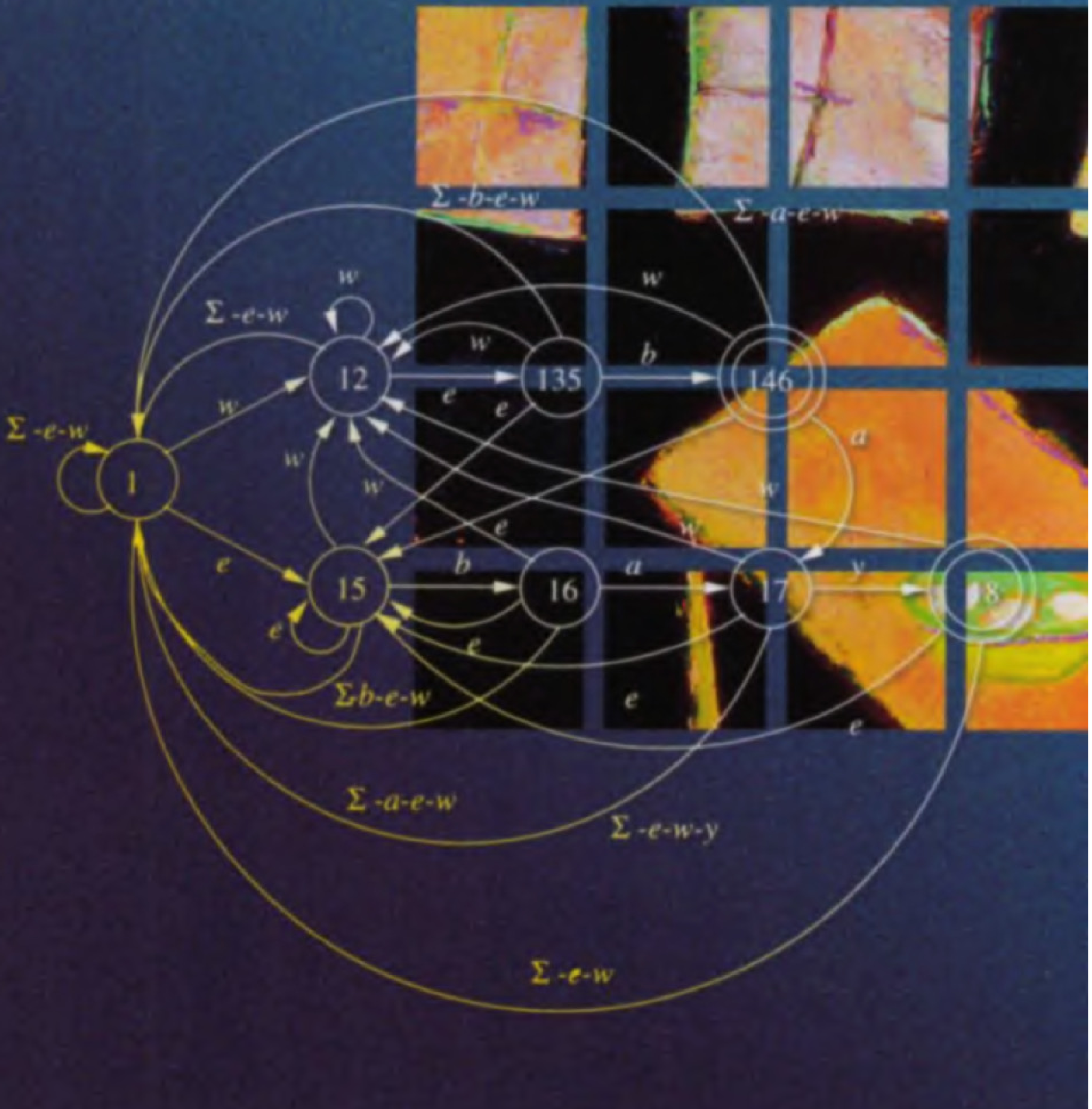
JOHN E. HOPCROFT

RAJEEV MOTWANI

JEFFREY D. ULLMAN

Introduction to Automata Theory, Languages, and Computation

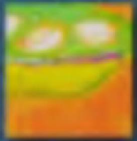


SECOND EDITION

Theory of Computation

Introduction to Automata Theory, Languages, and Computation

JOHN E. HOPCROFT • RAJEEV MOTWANI • JEFFREY D. ULLMAN



It has been more than 30 years since John Hopcroft and Jeffrey Ullman first published this classic book on formal languages, automata theory, and computational complexity. With this long-awaited revision, the authors continue to present the material in a concise and straightforward manner, now with an eye out for the practical applications along with the mathematics.

This edition has been revised to make it more accessible to today's students, including the addition of more material on writing proofs, more figures and pictures to convey ideas, sidebars to highlight related material, and a less formal writing style. It includes many new exercises in each chapter to help readers confirm and enhance their understanding of the material.

FEATURES

- Completely rewritten to be less formal, providing more accessibility to undergraduate students
- Emphasizes modern applications of the theory
- Uses numerous figures to help convey ideas
- Provides more detail and intuition for definitions and proofs
- Includes special sidebars to present supplemental material that may be of interest to readers
- Challenges readers with extensive exercises of wide-ranging difficulty levels
- Presents a graphical notation for PDA's and Turing machines.

John E. Hopcroft is the Joseph Silbert Dean of Engineering at Cornell University, and winner of the 1986 A. M. Turing Award.

Rajeev Motwani is Associate Professor and Director of Graduate Studies for Computer Science at Stanford University.

Jeffrey D. Ullman is the Stanford W. Ascherman Professor of Computer Science at Stanford University.

Pearson Education

9 780201 441246 90000
ISBN 0-201-44124-1

Senior Acquisitions Editor	*Maite Suarez-Rivas*
Project Editor	*Katherine Harutunian*
Executive Marketing Manager	*Michael Hirsch*
Cover Design	*Leslie Haimes*
Art Direction	*Regina Hagen*
Prepress and Manufacturing	*Caroline Fell*

Access the latest information about Addison-Wesley titles from our World Wide Web site: http://www.awl.com

Library of Congress Cataloging-in-Publication Data

Hopcroft, John E., 1939-
Introduction to automata theory, languages, and computation / John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman.—2nd ed.
p. cm.
ISBN 0-201-44124-1
1. Machine theory. 2. Formal languages. 3. Computational complexity. I. Motwani, Rajeev. II. Ullman, Jeffrey D., 1942-.

QA267 .H56 2001
511.3—dc21 00-064608

345678910-MA-04030201

Preface

In the preface from the 1979 predecessor to this book, Hopcroft and Ullman marveled at the fact that the subject of automata had exploded, compared with its state at the time they wrote their first book, in 1969. Truly, the 1979 book contained many topics not found in the earlier work and was about twice its size. If you compare this book with the 1979 book, you will find that, like the automobiles of the 1970's, this book is "larger on the outside, but smaller on the inside." That sounds like a retrograde step, but we are happy with the changes for several reasons.

First, in 1979, automata and language theory was still an area of active research. A purpose of that book was to encourage mathematically inclined students to make new contributions to the field. Today, there is little direct research in automata theory (as opposed to its applications), and thus little motivation for us to retain the succinct, highly mathematical tone of the 1979 book.

Second, the role of automata and language theory has changed over the past two decades. In 1979, automata was largely a graduate-level subject, and we imagined our reader was an advanced graduate student, especially those using the later chapters of the book. Today, the subject is a staple of the undergraduate curriculum. As such, the content of the book must assume less in the way of prerequisites from the student, and therefore must provide more of the background and details of arguments than did the earlier book.

A third change in the environment is that Computer Science has grown to an almost unimaginable degree in the past two decades. While in 1979 it was often a challenge to fill up a curriculum with material that we felt would survive the next wave of technology, today very many subdisciplines compete for the limited amount of space in the undergraduate curriculum.

Fourthly, CS has become a more vocational subject, and there is a severe pragmatism among many of its students. We continue to believe that aspects of automata theory are essential tools in a variety of new disciplines, and we believe that the theoretical, mind-expanding exercises embodied in the typical automata course retain their value, no matter how much the student prefers to learn only the most immediately monetizable technology. However, to assure a continued place for the subject on the menu of topics available to the computer science student, we believe it is necessary to emphasize the applications along with the mathematics. Thus, we have replaced a number of the more abstruse topics in the earlier book with examples of how the ideas are used today. While applications of automata and language theory to compilers are now so well understood that they are normally covered in a compiler course, there are a variety of more recent uses, including model-checking algorithms to verify protocols and document-description languages that are patterned on context-free grammars.

A final explanation for the simultaneous growth and shrinkage of the book is that we were today able to take advantage of the TEX and LATEX typesetting systems developed by Don Knuth and Les Lamport. The latter, especially, encourages the "open" style of typesetting that makes books larger, but easier to read. We appreciate the efforts of both men.

Use of the Book

This book is suitable for a quarter or semester course at the Junior level or above. At Stanford, we have used the notes in CS154, the course in automata and language theory. It is a one-quarter course, which both Rajeev and Jeff have taught. Because of the limited time available, Chapter 11 is not covered, and some of the later material, such as the more difficult polynomial-time reductions in Section 10.4 are omitted as well. The book's Web site (see below) includes notes and syllabi for several offerings of CS154.

Some years ago, we found that many graduate students came to Stanford with a course in automata theory that did not include the theory of intractability. As the Stanford faculty believes that these ideas are essential for every computer scientist to know at more than the level of "NP-complete means it takes too long," there is another course, CS154N, that students may take to cover only Chapters 8, 9, and 10. They actually participate in roughly the last third of CS154 to fulfill the CS154N requirement. Even today, we find several students each quarter availing themselves of this option. Since it requires little extra effort, we recommend the approach.

Prerequisites

To make best use of this book, students should have taken previously a course covering discrete mathematics, e.g., graphs, trees, logic, and proof techniques. We assume also that they have had several courses in programming, and are familiar with common data structures, recursion, and the role of major system components such as compilers. These prerequisites should be obtained in a typical freshman-sophomore CS program.

Exercises

The book contains extensive exercises, with some for almost every section. We indicate harder exercises or parts of exercises with an exclamation point. The hardest exercises have a double exclamation point.

Some of the exercises or parts are marked with a star. For these exercises, we shall endeavor to maintain solutions accessible through the book's Web page. These solutions are publicly available and should be used for self-testing. Note that in a few cases, one exercise B asks for modification or adaptation of your solution to another exercise A. If certain parts of A have solutions, then you should expect the corresponding parts of B to have solutions as well.

Support on the World Wide Web

The book's home page is

```
http://www-db.stanford.edu/~ullman/ialc.html
```

Here are solutions to starred exercises, errata as we learn of them, and backup materials. We hope to make available the notes for each offering of CS154 as we teach it, including homeworks, solutions, and exams.

Acknowledgements

A handout on "how to do proofs" by Craig Silverstein influenced some of the material in Chapter 1. Comments and errata on drafts of this book were received from: Zoe Abrams, George Candea, Haowen Chen, Byong-Gun Chun, Jeffrey Shallit, Bret Taylor, Jason Townsend, and Erik Uzureau. They are gratefully acknowledged. Remaining errors are ours, of course.

J. E. H.
R. M.
J. D. U.
Ithaca NY and Stanford CA
September, 2000

Table of Contents

Chapter 1

Automata: The Methods and the Madness

Automata theory is the study of abstract computing devices, or "machines." Before there were computers, in the 1930's, A. Turing studied an abstract machine that had all the capabilities of today's computers, at least as far as in what they could compute. Turing's goal was to describe precisely the boundary between what a computing machine could do and what it could not do; his conclusions apply not only to his abstract *Turing machines*, but to today's real machines.

In the 1940's and 1950's, simpler kinds of machines, which we today call "finite automata," were studied by a number of researchers. These automata, originally proposed to model brain function, turned out to be extremely useful for a variety of other purposes, which we shall mention in Section 1.1. Also in the late 1950's, the linguist N. Chomsky began the study of formal "grammars." While not strictly machines, these grammars have close relationships to abstract automata and serve today as the basis of some important software components, including parts of compilers.

In 1969, S. Cook extended Turing's study of what could and what could not be computed. Cook was able to separate those problems that can be solved efficiently by computer from those problems that can in principle be solved, but in practice take so much time that computers are useless for all but very small instances of the problem. The latter class of problems is called "intractable," or "NP-hard." It is highly unlikely that even the exponential improvement in computing speed that computer hardware has been following ("Moore's Law") will have significant impact on our ability to solve large instances of intractable problems.

All of these theoretical developments bear directly on what computer scientists do today. Some of the concepts, like finite automata and certain kinds of formal grammars, are used in the design and construction of important kinds of software. Other concepts, like the Turing machine, help us understand what we can expect from our software. Especially, the theory of intractable problems lets us deduce whether we are likely to be able to meet a problem "head-on" and write a program to solve it (because it is not in the intractable class), or whether we have to find some way to work around the intractable problem: find an approximation, use a heuristic, or use some other method to limit the amount of time the program will spend solving the problem.

In this introductory chapter, we begin with a very high-level view of what automata theory is about, and what its uses are. Much of the chapter is devoted to a survey of proof techniques and tricks for discovering proofs. We cover deductive proofs, reformulating statements, proofs by contradiction, proofs by induction, and other important concepts. A final section introduces the concepts that pervade automata theory: alphabets, strings, and languages.

1.1 Why Study Automata Theory?

There are several reasons why the study of automata and complexity is an important part of the core of Computer Science. This section serves to introduce the reader to the principal motivation and also outlines the major topics covered in this book.

1.1.1 Introduction to Finite Automata

Finite automata are a useful model for many important kinds of hardware and software. We shall see, starting in Chapter 2, examples of how the concepts are used. For the moment, let us just list some of the most important kinds:

1. Software for designing and checking the behavior of digital circuits.

2. The "lexical analyzer" of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords, and punctuation.

3. Software for scanning large bodies of text, such as collections of Web pages, to find occurrences of words, phrases, or other patterns.

4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange of information.

While we shall soon meet a precise definition of automata of various types, let us begin our informal introduction with a sketch of what a finite automaton is and does. There are many systems or components, such as those enumerated above, that may be viewed as being at all times in one of a finite number of "states." The purpose of a state is to remember the relevant portion of the system's history. Since there are only a finite number of states, the entire history generally cannot be remembered, so the system must be designed carefully, to remember what is important and forget what is not. The advantage of having only a finite number of states is that we can implement the system with a fixed set of resources. For example, we could implement it in hardware as a circuit, or as a simple form of program that can make decisions looking only at a limited amount of data or using the position in the code itself to make the decision.

Example 1.1: Perhaps the simplest nontrivial finite automaton is an on/off switch. The device remembers whether it is in the "on" state or the "off" state, and it allows the user to press a button whose effect is different, depending on the state of the switch. That is, if the switch is in the off state, then pressing the button changes it to the on state, and if the switch is in the on state, then pressing the same button turns it to the off state.

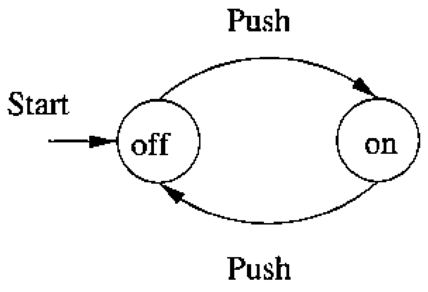


Figure 1.1: A finite automaton modeling an on/off switch

The finite-automaton model for the switch is shown in Fig. 1.1. As for all finite automata, the states are represented by circles; in this example, we have named the states *on* and *off*. Arcs between states are labeled by "inputs," which represent external influences on the system. Here, both arcs are labeled by the input *Push*, which represents a user pushing the button. The intent of the two arcs is that whichever state the system is in, when the *Push* input is received it goes to the other state.

One of the states is designated the "start state," the state in which the system is placed initially. In our example, the start state is *off*, and we conventionally indicate the start state by the word *Start* and an arrow leading to that state.

It is often necessary to indicate one or more states as "final" or "accepting" states. Entering one of these states after a sequence of inputs indicates that the input sequence is good in some way. For instance, we could have regarded the state *on* in Fig. 1.1 as accepting, because in that state, the device being controlled by the switch will operate. It is conventional to designate accepting states by a double circle, although we have not made any such designation in Fig. 1.1. □

Example 1.2: Sometimes, what is remembered by a state can be much more complex than an on/off choice. Figure 1.2 shows another finite automaton that could be part of a lexical analyzer. The job of this automaton is to recognize the keyword **then**. It thus needs five states, each of which represents a different position in the word **then** that has been reached so far. These positions correspond to the prefixes of the word, ranging from the empty string (i.e., nothing of the word has been seen so far) to the complete word.

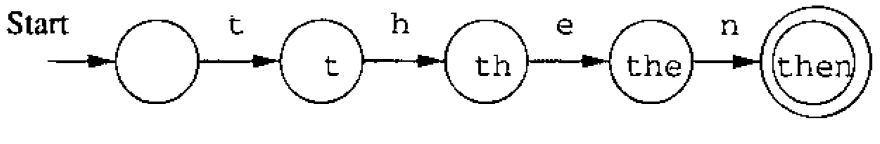


Figure 1.2: A finite automaton modeling recognition of **then**

In Fig. 1.2, the five states are named by the prefix of **then** seen so far. Inputs correspond to letters. We may imagine that the lexical analyzer examines one character of the program that it is compiling at a time, and the next character to be examined is the input to the automaton. The start state corresponds to the empty string, and each state has a transition on the next letter of **then** to the state that corresponds to the next-larger prefix. The state named **then** is entered when the input has spelled the word **then**. Since it is the job of this automaton to recognize when **then** has been seen, we could consider that state the lone accepting state. □

1.1.2 Structural Representations

There are two important notations that are not automaton-like, but play an important role in the study of automata and their applications.

1. *Grammars* are useful models when designing software that processes data with a recursive structure. The best-known example is a "parser," the component of a compiler that deals with the recursively nested features of the typical programming language, such as expressions — arithmetic, conditional, and so on. For instance, a grammatical rule like $E \Rightarrow E + E$ states that an expression can be formed by taking any two expressions and connecting them by a plus sign; this rule is typical of how expressions of real programming languages are formed. We introduce context-free grammars, as they are usually called, in Chapter 5.

2. *Regular Expressions* also denote the structure of data, especially text strings. As we shall see in Chapter 3, the patterns of strings they describe are exactly the same as what can be described by finite automata. The style of these expressions differs significantly from that of grammars, and we shall content ourselves with a simple example here. The UNIX-style regular expression `'[A-Z][a-z]*[ ][A-Z][A-Z]'` represents capitalized words followed by a space and two capital letters. This expression represents patterns in text that could be a city and state, e.g., `Ithaca NY`. It misses multiword city names, such as `Palo Alto CA`, which could be captured by the more complex expression

```
'([A-Z][a-z]*[ ])*[ ][A-Z][A-Z]'
```

When interpreting such expressions, we only need to know that [A-Z] represents a range of characters from capital "A" to capital "Z" (i.e., any capital letter), and [ ] is used to represent the blank character alone. Also, the symbol * represents "any number of" the preceding expression. Parentheses are used to group components of the expression; they do not represent characters of the text described.

1.1.3 Automata and Complexity

Automata are essential for the study of the limits of computation. As we mentioned in the introduction to the chapter, there are two important issues:

1. What can a computer do at all? This study is called "decidability," and the problems that can be solved by computer are called "decidable." This topic is addressed in Chapter 9.

2. What can a computer do efficiently? This study is called "intractability," and the problems that can be solved by a computer using no more time than some slowly growing function of the size of the input are called "tractable." Often, we take all polynomial functions to be "slowly growing," while functions that grow faster than any polynomial are deemed to grow too fast. The subject is studied in Chapter 10.

1.2 Introduction to Formal Proof

If you studied plane geometry in high school any time before the 1990's, you most likely had to do some detailed "deductive proofs," where you showed the truth of a statement by a detailed sequence of steps and reasons. While geometry has its practical side (e.g., you need to know the rule for computing the area of a rectangle if you need to buy the correct amount of carpet for a room), the study of formal proof methodologies was at least as important a reason for covering this branch of mathematics in high school.

In the USA of the 1990's it became popular to teach proof as a matter of personal feelings about the statement. While it is good to feel the truth of a statement you need to use, important techniques of proof are no longer mastered in high school. Yet proof is something that every computer scientist needs to understand. Some computer scientists take the extreme view that a formal proof of the correctness of a program should go hand-in-hand with the writing of the program itself. We doubt that doing so is productive. On the other hand, there are those who say that proof has no place in the discipline of programming. The slogan "if you are not sure your program is correct, run it and see" is commonly offered by this camp.

Our position is between these two extremes. Testing programs is surely essential. However, testing goes only so far, since you cannot try your program on every input. More importantly, if your program is complex — say a tricky recursion or iteration — then if you don't understand what is going on as you go around a loop or call a function recursively, it is unlikely that you will write the code correctly. When your testing tells you the code is incorrect, you still need to get it right.

To make your iteration or recursion correct, you need to set up an inductive hypothesis, and it is helpful to reason, formally or informally, that the hypothesis is consistent with the iteration or recursion. This process of understanding the workings of a correct program is essentially the same as the process of proving theorems by induction. Thus, in addition to giving you models that are useful for certain types of software, it has become traditional for a course on automata theory to cover methodologies of formal proof. Perhaps more than other core subjects of computer science, automata theory lends itself to natural and interesting proofs, both of the ***deductive*** kind (a sequence of justified steps) and the ***inductive*** kind (recursive proofs of a parameterized statement that use the statement itself with "lower" values of the parameter).

1.2.1 Deductive Proofs

As mentioned above, a deductive proof consists of a sequence of statements whose truth leads us from some initial statement, called the ***hypothesis*** or the ***given statement(s)***, to a ***conclusion*** statement. Each step in the proof must follow, by some accepted logical principle, from either the given facts, or some of the previous statements in the deductive proof, or a combination of these.

The hypothesis may be true or false, typically depending on values of its parameters. Often, the hypothesis consists of several independent statements connected by a logical AND. In those cases, we talk of each of these statements as a hypothesis, or as a given statement.

The theorem that is proved when we go from a hypothesis H to a conclusion C is the statement "if H then C." We say that C is ***deduced*** from H. An example theorem of the form "if H then C" will illustrate these points.

Theorem 1.3: If $x \geq 4$, then $2^x \geq x^2$. □

It is not hard to convince ourselves informally that Theorem 1.3 is true, although a formal proof requires induction and will be left for Example 1.17. First, notice that the hypothesis H is "$x \geq 4$." This hypothesis has a parameter, x, and thus is neither true nor false. Rather, its truth depends on the value of the parameter x; e.g., H is true for $x = 6$ and false for $x = 2$.

Likewise, the conclusion C is "$2^x \geq x^2$." This statement also uses parameter x and is true for certain values of x and not others. For example, C is false for $x = 3$, since $2^3 = 8$, which is not as large as $3^2 = 9$. On the other hand, C is true for $x = 4$, since $2^4 = 4^2 = 16$. For $x = 5$, the statement is also true, since $2^5 = 32$ is at least as large as $5^2 = 25$.

Perhaps you can see the intuitive argument that tells us the conclusion $2^x \geq x^2$ will be true whenever $x \geq 4$. We already saw that it is true for $x = 4$. As x grows larger than 4, the left side, 2^x doubles each time x increases by 1. However, the right side, x^2, grows by the ratio $\left(\frac{x+1}{x}\right)^2$. If $x \geq 4$, then $(x+1)/x$ cannot be greater than 1.25, and therefore $\left(\frac{x+1}{x}\right)^2$ cannot be bigger than 1.5625. Since $1.5625 < 2$, each time x increases above 4 the left side 2^x grows more than the right side x^2. Thus, as long as we start from a value like $x = 4$ where the inequality $2^x \geq x^2$ is already satisfied, we can increase x as much as we like, and the inequality will still be satisfied.

We have now completed an informal but accurate proof of Theorem 1.3. We shall return to the proof and make it more precise in Example 1.17, after we introduce "inductive" proofs.

Theorem 1.3, like all interesting theorems, involves an infinite number of related facts, in this case the statement "if $x \geq 4$ then $2^x \geq x^2$" for all integers x. In fact, we do not need to assume x is an integer, but the proof talked about repeatedly increasing x by 1, starting at $x = 4$, so we really addressed only the situation where x is an integer.

Theorem 1.3 can be used to help deduce other theorems. In the next example, we consider a complete deductive proof of a simple theorem that uses Theorem 1.3.

Theorem 1.4: If x is the sum of the squares of four positive integers, then $2^x \geq x^2$.

PROOF: The intuitive idea of the proof is that if the hypothesis is true for x, that is, x is the sum of the squares of four positive integers, then x must be at least 4. Therefore, the hypothesis of Theorem 1.3 holds, and since we believe that theorem, we may state that its conclusion is also true for x. The reasoning can be expressed as a sequence of steps. Each step is either the hypothesis of the theorem to be proved, part of that hypothesis, or a statement that follows from one or more previous statements.

By "follows" we mean that if the hypothesis of some theorem is a previous statement, then the conclusion of that theorem is true, and can be written down as a statement of our proof. This logical rule is often called *modus ponens*; i.e., if we know H is true, and we know "if H then C" is true, we may conclude that C is true. We also allow certain other logical steps to be used in creating a statement that follows from one or more previous statements. For instance, if A and B are two previous statements, then we can deduce and write down the statement "A and B."

Figure 1.3 shows the sequence of statements we need to prove Theorem 1.4. While we shall not generally prove theorems in such a stylized form, it helps to think of proofs as very explicit lists of statements, each with a precise justification. In step (1), we have repeated one of the given statements of the theorem: that x is the sum of the squares of four integers. It often helps in proofs if we name quantities that are referred to but not named, and we have done so here, giving the four integers the names a, b, c, and d.

	Statement	Justification
1.	$x = a^2 + b^2 + c^2 + d^2$	Given
2.	$a \geq 1$; $b \geq 1$; $c \geq 1$; $d \geq 1$	Given
3.	$a^2 \geq 1$; $b^2 \geq 1$; $c^2 \geq 1$; $d^2 \geq 1$	(2) and properties of arithmetic
4.	$x \geq 4$	(1), (3), and properties of arithmetic
5.	$2^x \geq x^2$	(4) and Theorem 1.3

Figure 1.3: A formal proof of Theorem 1.4

In step (2), we put down the other part of the hypothesis of the theorem: that the values being squared are each at least 1. Technically, this statement represents four distinct statements, one for each of the four integers involved. Then, in step (3) we observe that if a number is at least 1, then its square is also at least 1. We use as a justification the fact that statement (2) holds, and "properties of arithmetic." That is, we assume the reader knows, or can prove simple statements about how inequalities work, such as the statement "if $y \geq 1$, then $y^2 \geq 1$."

Step (4) uses statements (1) and (3). The first statement tells us that x is the sum of the four squares in question, and statement (3) tells us that each of the squares is at least 1. Again using well-known properties of arithmetic, we conclude that x is at least $1 + 1 + 1 + 1$, or 4.

At the final step (5), we use statement (4), which is the hypothesis of Theorem 1.3. The theorem itself is the justification for writing down its conclusion, since its hypothesis is a previous statement. Since the statement (5) that is the conclusion of Theorem 1.3 is also the conclusion of Theorem 1.4, we have now proved Theorem 1.4. That is, we have started with the hypothesis of that theorem, and have managed to deduce its conclusion. $\square$

1.2.2 Reduction to Definitions

In the previous two theorems, the hypotheses used terms that should have been familiar: integers, addition, and multiplication, for instance. In many other theorems, including many from automata theory, the terms used in the statement may have implications that are less obvious. A useful way to proceed in many proofs is:

- If you are not sure how to start a proof, convert all terms in the hypothesis to their definitions.

Here is an example of a theorem that is simple to prove once we have expressed its statement in elementary terms. It uses the following two definitions:

1. A set S is *finite* if there exists an integer n such that S has exactly n elements. We write $\|S\| = n$, where $\|S\|$ is used to denote the number

of elements in a set S. If the set S is not finite, we say S is *infinite*. Intuitively, an infinite set is a set that contains more than any integer number of elements.

2. If S and T are both subsets of some set U, then T is the *complement* of S (with respect to U) if $S \cup T = U$ and $S \cap T = \emptyset$. That is, each element of U is in exactly one of S and T; put another way, T consists of exactly those elements of U that are not in S.

Theorem 1.5: Let S be a finite subset of some infinite set U. Let T be the complement of S with respect to U. Then T is infinite.

PROOF: Intuitively, this theorem says that if you have an infinite supply of something (U), and you take a finite amount away (S), then you still have an infinite amount left. Let us begin by restating the facts of the theorem as in Fig. 1.4.

Original Statement	New Statement
S is finite	There is a integer n such that $\|S\| = n$
U is infinite	For no integer p is $\|U\| = p$
T is the complement of S	$S \cup T = U$ and $S \cap T = \emptyset$

Figure 1.4: Restating the givens of Theorem 1.5

We are still stuck, so we need to use a common proof technique called "proof by contradiction." In this proof method, to be discussed further in Section 1.3.3, we assume that the conclusion is false. We then use that assumption, together with parts of the hypothesis, to prove the opposite of one of the given statements of the hypothesis. We have then shown that it is impossible for all parts of the hypothesis to be true and for the conclusion to be false at the same time. The only possibility that remains is for the conclusion to be true whenever the hypothesis is true. That is, the theorem is true.

In the case of Theorem 1.5, the contradiction of the conclusion is "T is finite." Let us assume T is finite, along with the statement of the hypothesis that says S is finite; i.e., $\|S\| = n$ for some integer n. Similarly, we can restate the assumption that T is finite as $\|T\| = m$ for some integer m.

Now one of the given statements tells us that $S \cup T = U$, and $S \cap T = \emptyset$. That is, the elements of U are exactly the elements of S and T. Thus, there must be $n + m$ elements of U. Since $n + m$ is an integer, and we have shown $\|U\| = n + m$, it follows that U is finite. More precisely, we showed the number of elements in U is some integer, which is the definition of "finite." But the statement that U is finite contradicts the given statement that U is infinite. We have thus used the contradiction of our conclusion to prove the contradiction of one of the given statements of the hypothesis, and by the principle of "proof by contradiction" we may conclude the theorem is true. □

Proofs do not have to be so wordy. Having seen the ideas behind the proof, let us reprove the theorem in a few lines.

PROOF: (of Theorem 1.5) We know that $S \cup T = U$ and S and T are disjoint, so $\|S\| + \|T\| = \|U\|$. Since S is finite, $\|S\| = n$ for some integer n, and since U is infinite, there is no integer p such that $\|U\| = p$. So assume that T is finite; that is, $\|T\| = m$ for some integer m. Then $\|U\| = \|S\| + \|T\| = n + m$, which contradicts the given statement that there is no integer p equal to $\|U\|$. □

1.2.3 Other Theorem Forms

The "if-then" form of theorem is most common in typical areas of mathematics. However, we see other kinds of statements proved as theorems also. In this section, we shall examine the most common forms of statement and what we usually need to do to prove them.

Ways of Saying "If-Then"

First, there are a number of kinds of theorem statements that look different from a simple "if H then C" form, but are in fact saying the same thing: if hypothesis H is true for a given value of the parameter(s), then the conclusion C is true for the same value. Here are some of the other ways in which "if H then C" might appear.

1. H implies C.
2. H only if C.
3. C if H.
4. Whenever H holds, C follows.

We also see many variants of form (4), such as "if H holds, then C follows," or "whenever H holds, C holds."

Example 1.6: The statement of Theorem 1.3 would appear in these four forms as:

1. $x \geq 4$ implies $2^x \geq x^2$.
2. $x \geq 4$ only if $2^x \geq x^2$.
3. $2^x \geq x^2$ if $x \geq 4$.
4. Whenever $x \geq 4$, $2^x \geq x^2$ follows.

□

Statements With Quantifiers

Many theorems involve statements that use the *quantifiers* "for all" and "there exists," or similar variations, such as "for every" instead of "for all." The order in which these quantifiers appear affects what the statement means. It is often helpful to see statements with more than one quantifier as a "game" between two players — for-all and there-exists — who take turns specifying values for the parameters mentioned in the theorem. "For-all" must consider all possible choices, so for-all's choices are generally left as variables. However, "there-exists" only has to pick one value, which may depend on the values picked by the players previously. The order in which the quantifiers appear in the statement determines who goes first. If the last player to make a choice can always find some allowable value, then the statement is true.

For example, consider an alternative definition of "infinite set": set S is *infinite* if and only if for all integers n, there exists a subset T of S with exactly n members. Here, "for-all" precedes "there-exists," so we must consider an arbitrary integer n. Now, "there-exists" gets to pick a subset T, and may use the knowledge of n to do so. For instance, if S were the set of integers, "there-exists" could pick the subset $T = \{1, 2, \ldots, n\}$ and thereby succeed regardless of n. That is a proof that the set of integers is infinite.

The following statement looks like the definition of "infinite," but is *incorrect* because it reverses the order of the quantifiers: "there exists a subset T of set S such that for all n, set T has exactly n members." Now, given a set S such as the integers, player "there-exists" can pick any set T; say $\{1, 2, 5\}$ is picked. For this choice, player "for-all" must show that T has n members for *every* possible n. However, "for-all" cannot do so. For instance, it is false for $n = 4$, or in fact for any $n \neq 3$.

In addition, in formal logic one often sees the operator $\rightarrow$ in place of "if-then." That is, the statement "if H then C" could appear as $H \rightarrow C$ in some mathematical literature; we shall not use it here.

If-And-Only-If Statements

Sometimes, we find a statement of the form "A if and only if B." Other forms of this statement are "A iff B,"[1] "A is equivalent to B," or "A exactly when B." This statement is actually two if-then statements: "if A then B," and "if B then A." We prove "A if and only if B" by proving these two statements:

[1] Iff, short for "if and only if," is a non-word that is used in some mathematical treatises for succinctness.

How Formal Do Proofs Have to Be?

The answer to this question is not easy. The bottom line regarding proofs is that their purpose is to convince someone, whether it is a grader of your classwork or yourself, about the correctness of a strategy you are using in your code. If it is convincing, then it is enough; if it fails to convince the "consumer" of the proof, then the proof has left out too much.

Part of the uncertainty regarding proofs comes from the different knowledge that the consumer may have. Thus, in Theorem 1.4, we assumed you knew all about arithmetic, and would believe a statement like "if $y \geq 1$ then $y^2 \geq 1$." If you were not familiar with arithmetic, we would have to prove that statement by some steps in our deductive proof.

However, there are certain things that are required in proofs, and omitting them surely makes the proof inadequate. For instance, any deductive proof that uses statements which are not justified by the given or previous statements, cannot be adequate. When doing a proof of an "if and only if" statement, we must surely have one proof for the "if" part and another proof for the "only-if" part. As an additional example, inductive proofs (discussed in Section 1.4) require proofs of the basis and induction parts.

1. The *if part*: "if B then A," and

2. The *only-if part*: "if A then B," which is often stated in the equivalent form "A only if B."

The proofs can be presented in either order. In many theorems, one part is decidedly easier than the other, and it is customary to present the easy direction first and get it out of the way.

In formal logic, one may see the operator $\leftrightarrow$ or $\equiv$ to denote an "if-and-only-if" statement. That is, $A \equiv B$ and $A \leftrightarrow B$ mean the same as "A if and only if B."

When proving an if-and-only-if statement, it is important to remember that you must prove both the "if" and "only-if" parts. Sometimes, you will find it helpful to break an if-and-only-if into a succession of several equivalences. That is, to prove "A if and only if B," you might first prove "A if and only if C," and then prove "C if and only if B." That method works, as long as you remember that each if-and-only-if step must be proved in both directions. Proving any one step in only one of the directions invalidates the entire proof.

The following is an example of a simple if-and-only-if proof. It uses the notations:

1. $\lfloor x \rfloor$, the *floor* of real number x, is the greatest integer equal to or less than x.

2. $\lceil x \rceil$, the *ceiling* of real number x, is the least integer equal to or greater than x.

Theorem 1.7: Let x be a real number. Then $\lfloor x \rfloor = \lceil x \rceil$ if and only if x is an integer.

PROOF: (Only-if part) In this part, we assume $\lfloor x \rfloor = \lceil x \rceil$ and try to prove x is an integer. Using the definitions of the floor and ceiling, we notice that $\lfloor x \rfloor \leq x$, and $\lceil x \rceil \geq x$. However, we are given that $\lfloor x \rfloor = \lceil x \rceil$. Thus, we may substitute the floor for the ceiling in the first inequality to conclude $\lceil x \rceil \leq x$. Since both $\lceil x \rceil \leq x$ and $\lceil x \rceil \geq x$ hold, we may conclude by properties of arithmetic inequalities that $\lceil x \rceil = x$. Since $\lceil x \rceil$ is always an integer, x must also be an integer in this case.

(If part) Now, we assume x is an integer and try to prove $\lfloor x \rfloor = \lceil x \rceil$. This part is easy. By the definitions of floor and ceiling, when x is an integer, both $\lfloor x \rfloor$ and $\lceil x \rceil$ are equal to x, and therefore equal to each other. □

1.2.4 Theorems That Appear Not to Be If-Then Statements

Sometimes, we encounter a theorem that appears not to have a hypothesis. An example is the well-known fact from trigonometry:

Theorem 1.8: $\sin^2 \theta + \cos^2 \theta = 1$. □

Actually, this statement *does* have a hypothesis, and the hypothesis consists of all the statements you need to know to interpret the statement. In particular, the hidden hypothesis is that θ is an angle, and therefore the functions sine and cosine have their usual meaning for angles. From the definitions of these terms, and the Pythagorean Theorem (in a right triangle, the square of the hypotenuse equals the sum of the squares of the other two sides), you could prove the theorem. In essence, the if-then form of the theorem is really: "if θ is an angle, then $\sin^2 \theta + \cos^2 \theta = 1$."

1.3 Additional Forms of Proof

In this section, we take up several additional topics concerning how to construct proofs:

1. Proofs about sets.

2. Proofs by contradiction.

3. Proofs by counterexample.

1.3.1 Proving Equivalences About Sets

In automata theory, we are frequently asked to prove a theorem which says that the sets constructed in two different ways are the same sets. Often, these sets are sets of character strings, and the sets are called "languages," but in this section the nature of the sets is unimportant. If E and F are two expressions representing sets, the statement $E = F$ means that the two sets represented are the same. More precisely, every element in the set represented by E is in the set represented by F, and every element in the set represented by F is in the set represented by E.

Example 1.9: The ***commutative law of union*** says that we can take the union of two sets R and S in either order. That is, $R \cup S = S \cup R$. In this case, E is the expression $R \cup S$ and F is the expression $S \cup R$. The commutative law of union says that $E = F$. □

We can write a set-equality $E = F$ as an if-and-only-if statement: an element x is in E if and only if x is in F. As a consequence, we see the outline of a proof of any statement that asserts the equality of two sets $E = F$; it follows the form of any if-and-only-if proof:

1. Proof that if x is in E, then x is in F.

2. Prove that if x is in F, then x is in E.

As an example of this proof process, let us prove the ***distributive law of union over intersection***:

Theorem 1.10: $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

PROOF: The two set-expressions involved are $E = R \cup (S \cap T)$ and

$$F = (R \cup S) \cap (R \cup T)$$

We shall prove the two parts of the theorem in turn. In the "if" part we assume element x is in E and show it is in F. This part, summarized in Fig. 1.5, uses the definitions of union and intersection, with which we assume you are familiar.

Then, we must prove the "only-if" part of the theorem. Here, we assume x is in F and show it is in E. The steps are summarized in Fig. 1.6. Since we have now proved both parts of the if-and-only-if statement, the distributive law of union over intersection is proved. □

1.3.2 The Contrapositive

Every if-then statement has an equivalent form that in some circumstances is easier to prove. The ***contrapositive*** of the statement "if H then C" is "if not C then not H." A statement and its contrapositive are either both true or both false, so we can prove either to prove the other.

To see why "if H then C" and "if not C then not H" are logically equivalent, first observe that there are four cases to consider:

	Statement	Justification
1.	x is in $R \cup (S \cap T)$	Given
2.	x is in R or x is in $S \cap T$	(1) and definition of union
3.	x is in R or x is in both S and T	(2) and definition of intersection
4.	x is in $R \cup S$	(3) and definition of union
5.	x is in $R \cup T$	(3) and definition of union
6.	x is in $(R \cup S) \cap (R \cup T)$	(4), (5), and definition of intersection

Figure 1.5: Steps in the "if" part of Theorem 1.10

	Statement	Justification
1.	x is in $(R \cup S) \cap (R \cup T)$	Given
2.	x is in $R \cup S$	(1) and definition of intersection
3.	x is in $R \cup T$	(1) and definition of intersection
4.	x is in R or x is in both S and T	(2), (3), and reasoning about unions
5.	x is in R or x is in $S \cap T$	(4) and definition of intersection
6.	x is in $R \cup (S \cap T)$	(5) and definition of union

Figure 1.6: Steps in the "only-if" part of Theorem 1.10

1. H and C both true.

2. H true and C false.

3. C true and H false.

4. H and C both false.

There is only one way to make an if-then statement false; the hypothesis must be true and the conclusion false, as in case (2). For the other three cases, including case (4) where the conclusion is false, the if-then statement itself is true.

Now, consider for which cases the contrapositive "if not C then not H" is false. In order for this statement to be false, its hypothesis (which is "not C") must be true, and its conclusion (which is "not H") must be false. But "not C" is true exactly when C is false, and "not H" is false exactly when H is true. These two conditions are again case (2), which shows that in each of the four cases, the original statement and its contrapositive are either both true or both false; i.e., they are logically equivalent.

Saying "If-And-Only-If" for Sets

As we mentioned, theorems that state equivalences of expressions about sets are if-and-only-if statements. Thus, Theorem 1.10 could have been stated: an element x is in $R \cup (S \cap T)$ if and only if x is in

$$(R \cup S) \cap (R \cup T)$$

Another common expression of a set-equivalence is with the locution "all-and-only." For instance, Theorem 1.10 could as well have been stated "the elements of $R \cup (S \cap T)$ are all and only the elements of

$$(R \cup S) \cap (R \cup T)$$

The Converse

Do not confuse the terms "contrapositive" and "converse." The ***converse*** of an if-then statement is the "other direction"; that is, the converse of "if H then C" is "if C then H." Unlike the contrapositive, which is logically equivalent to the original, the converse is *not* equivalent to the original statement. In fact, the two parts of an if-and-only-if proof are always some statement and its converse.

Example 1.11 : Recall Theorem 1.3, whose statement was: "if $x \geq 4$, then $2^x \geq x^2$." The contrapositive of this statement is "if not $2^x \geq x^2$ then not $x \geq 4$." In more colloquial terms, making use of the fact that "not $a \geq b$" is the same as $a < b$, the contrapositive is "if $2^x < x^2$ then $x < 4$." □

When we are asked to prove an if-and-only-if theorem, the use of the contrapositive in one of the parts allows us several options. For instance, suppose we want to prove the set equivalence $E = F$. Instead of proving "if x is in E then x is in F and if x is in F then x is in E," we could also put one direction in the contrapositive. One equivalent proof form is:

- If x is in E then x is in F, and if x is not in E then x is not in F.

We could also interchange E and F in the statement above.

1.3.3 Proof by Contradiction

Another way to prove a statement of the form "if H then C" is to prove the statement

- "H and not C implies falsehood."

That is, start by assuming both the hypothesis H and the negation of the conclusion C. Complete the proof by showing that something known to be false follows logically from H and not C. This form of proof is called ***proof by contradiction***.

Example 1.12: Recall Theorem 1.5, where we proved the if-then statement with hypothesis H = "U is an infinite set, S is a finite subset of U, and T is the complement of S with respect to U." The conclusion C was "T is infinite." We proceeded to prove this theorem by contradiction. We assumed "not C"; that is, we assumed T was finite.

Our proof was to derive a falsehood from H and not C. We first showed from the assumptions that S and T are both finite, that U also must be finite. But since U is stated in the hypothesis H to be infinite, and a set cannot be both finite and infinite, we have proved the logical statement "false." In logical terms, we have both a proposition p (U is finite) and its negation, not p (U is infinite). We then use the fact that "p and not p" is logically equivalent to "false." □

To see why proofs by contradiction are logically correct, recall from Section 1.3.2 that there are four combinations of truth values for H and C. Only the second case, H true and C false, makes the statement "if H then C" false. By showing that H and not C leads to falsehood, we are showing that case 2 cannot occur. Thus, the only possible combinations of truth values for H and C are the three combinations that make "if H then C" true.

1.3.4 Counterexamples

In real life, we are not told to prove a theorem. Rather, we are faced with something that seems true — a strategy for implementing a program for example — and we need to decide whether or not the "theorem" is true. To resolve the question, we may alternately try to prove the theorem, and if we cannot, try to prove that its statement is false.

Theorems generally are statements about an infinite number of cases, perhaps all values of its parameters. Indeed, strict mathematical convention will only dignify a statement with the title "theorem" if it has an infinite number of cases; statements that have no parameters, or that apply to only a finite number of values of its parameter(s) are called ***observations***. It is sufficient to show that an alleged theorem is false in any one case in order to show it is not a theorem. The situation is analogous to programs, since a program is generally considered to have a bug if it fails to operate correctly for even one input on which it was expected to work.

It often is easier to prove that a statement is not a theorem than to prove it *is* a theorem. As we mentioned, if S is any statement, then the statement "S is not a theorem" is itself a statement without parameters, and thus can be regarded as an observation rather than a theorem. The following are two examples, first of an obvious nontheorem, and the second a statement that just misses being a theorem and that requires some investigation before resolving the question of whether it is a theorem or not.

Alleged Theorem 1.13: All primes are odd. (More formally, we might say: if integer x is a prime, then x is odd.)

DISPROOF: The integer 2 is a prime, but 2 is even. □

Now, let us discuss a "theorem" involving modular arithmetic. There is an essential definition that we must first establish. If a and b are positive integers, then a mod b is the remainder when a is divided by b, that is, the unique integer r between 0 and $b-1$ such that $a = qb + r$ for some integer q. For example, 8 mod 3 = 2, and 9 mod 3 = 0. Our first proposed theorem, which we shall determine to be false, is:

Alleged Theorem 1.14: There is no pair of integers a and b such that

$$a \bmod b = b \bmod a$$

□

When asked to do things with pairs of objects, such as a and b here, it is often possible to simplify the relationship between the two by taking advantage of symmetry. In this case, we can focus on the case where $a < b$, since if $b < a$ we can swap a and b and get the same equation as in Alleged Theorem 1.14. we must be careful, however, not to forget the third case, where $a = b$. This case turns out to be fatal to our proof attempts.

Let us assume $a < b$. Then a mod $b = a$, since in the definition of a mod b we have $q = 0$ and $r = a$. That is, when $a < b$ we have $a = 0 \times b + a$. But b mod $a < a$, since anything mod a is between 0 and $a - 1$. Thus, when $a < b$, b mod $a < a$ mod b, so a mod $b = b$ mod a is impossible. Using the argument of symmetry above, we also know that a mod $b \neq b$ mod a when $b < a$.

However, consider the third case: $a = b$. Since x mod $x = 0$ for any integer x, we *do* have a mod $b = b$ mod a if $a = b$. We thus have a disproof of the alleged theorem:

DISPROOF: (of Alleged Theorem 1.14) Let $a = b = 2$. Then

$$a \bmod b = b \bmod a = 0$$

□

In the process of finding the counterexample, we have in fact discovered the exact conditions under which the alleged theorem holds. Here is the correct version of the theorem, and its proof.

Theorem 1.15: a mod $b = b$ mod a if and only if $a = b$.

PROOF: (If part) Assume $a = b$. Then as we observed above, $x \bmod x = 0$ for any integer x. Thus, $a \bmod b = b \bmod a = 0$ whenever $a = b$.

(Only-if part) Now, assume $a \bmod b = b \bmod a$. The best technique is a proof by contradiction, so assume in addition the negation of the conclusion; that is, assume $a \neq b$. Then since $a = b$ is eliminated, we have only to consider the cases $a < b$ and $b < a$.

We already observed above that when $a < b$, we have $a \bmod b = a$ and $b \bmod a < a$. Thus, these statements, in conjunction with the hypothesis $a \bmod b = b \bmod a$ lets us derive a contradiction.

By symmetry, if $b < a$ then $b \bmod a = b$ and $a \bmod b < b$. We again derive a contradiction of the hypothesis, and conclude the only-if part is also true. We have now proved both directions and conclude that the theorem is true. $\square$

1.4 Inductive Proofs

There is a special form of proof, called "inductive," that is essential when dealing with recursively defined objects. Many of the most familiar inductive proofs deal with integers, but in automata theory, we also need inductive proofs about such recursively defined concepts as trees and expressions of various sorts, such as the regular expressions that were mentioned briefly in Section 1.1.2. In this section, we shall introduce the subject of inductive proofs first with "simple" inductions on integers. Then, we show how to perform "structural" inductions on any recursively defined concept.

1.4.1 Inductions on Integers

Suppose we are given a statement $S(n)$, about an integer n, to prove. One common approach is to prove two things:

1. The *basis*, where we show $S(i)$ for a particular integer i. Usually, $i = 0$ or $i = 1$, but there are examples where we want to start at some higher i, perhaps because the statement S is false for a few small integers.

2. The *inductive step*, where we assume $n \geq i$, where i is the basis integer, and we show that "if $S(n)$ then $S(n + 1)$."

Intuitively, these two parts should convince us that $S(n)$ is true for every integer n that is equal to or greater than the basis integer i. We can argue as follows. Suppose $S(n)$ were false for one or more of those integers. Then there would have to be a smallest value of n, say j, for which $S(j)$ is false, and yet $j \geq i$. Now j could not be i, because we prove in the basis part that $S(i)$ is true. Thus, j must be greater than i. We now know that $j - 1 \geq i$, and $S(j-1)$ is true.

However, we proved in the inductive part that if $n \geq i$, then $S(n)$ implies $S(n + 1)$. Suppose we let $n = j - 1$. Then we know from the inductive step that $S(j - 1)$ implies $S(j)$. Since we also know $S(j - 1)$, we can conclude $S(j)$.

We have assumed the negation of what we wanted to prove; that is, we assumed $S(j)$ was false for some $j \geq i$. In each case, we derived a contradiction, so we have a "proof by contradiction" that $S(n)$ is true for all $n \geq i$.

Unfortunately, there is a subtle logical flaw in the above reasoning. Our assumption that we can pick a least $j \geq i$ for which $S(j)$ is false depends on our believing the principle of induction in the first place. That is, the only way to prove that we can find such a j is to prove it by a method that is essentially an inductive proof. However, the "proof" discussed above makes good intuitive sense, and matches our understanding of the real world. Thus, we generally take as an integral part of our logical reasoning system:

- *The Induction Principle*: If we prove $S(i)$ and we prove that for all $n \geq i$, $S(n)$ implies $S(n+1)$, then we may conclude $S(n)$ for all $n \geq i$.

The following two examples illustrate the use of the induction principle to prove theorems about integers.

Theorem 1.16: For all $n \geq 0$:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

PROOF: The proof is in two parts: the basis and the inductive step; we prove each in turn.

BASIS: For the basis, we pick $n = 0$. It might seem surprising that the theorem even makes sense for $n = 0$, since the left side of Equation (1.1) is $\sum_{i=1}^{0}$ when $n = 0$. However, there is a general principle that when the upper limit of a sum (0 in this case) is less than the lower limit (1 here), the sum is over no terms and therefore the sum is 0. That is, $\sum_{i=1}^{0} i^2 = 0$.

The right side of Equation (1.1) is also 0, since $0 \times (0+1) \times (2 \times 0+1)/6 = 0$. Thus, Equation (1.1) is true when $n = 0$.

INDUCTION: Now, assume $n \geq 0$. We must prove the inductive step, that Equation (1.1) implies the same formula with $n+1$ substituted for n. The latter formula is

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \tag{1.2}$$

We may simplify Equations (1.1) and (1.2) by expanding the sums and products on the right sides. These equations become:

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6 \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.4}$$

We need to prove (1.4) using (1.3), since in the induction principle, these are statements $S(n+1)$ and $S(n)$, respectively. The "trick" is to break the sum to $n+1$ on the right of (1.4) into a sum to n plus the $(n+1)$st term. In that way, we can replace the sum to n by the left side of (1.3) and show that (1.4) is true. These steps are as follows:

$$\left(\sum_{i=1}^{n} i^2\right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.5}$$

$$(2n^3 + 3n^2 + n)/6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.6}$$

The final verification that (1.6) is true requires only simple polynomial algebra on the left side to show it is identical to the right side. □

Example 1.17 : In the next example, we prove Theorem 1.3 from Section 1.2.1. Recall this theorem states that if $x \geq 4$, then $2^x \geq x^2$. We gave an informal proof based on the idea that the ratio $x^2/2^x$ shrinks as x grows above 4. We can make the idea precise if we prove the statement $2^x \geq x^2$ by induction on x, starting with a basis of $x = 4$. Note that the statement is actually false for $x < 4$.

BASIS: If $x = 4$, then 2^x and x^2 are both 16. Thus, $2^4 \geq 4^2$ holds.

INDUCTION: Suppose for some $x \geq 4$ that $2^x \geq x^2$. With this statement as the hypothesis, we need to prove the same statement, with $x+1$ in place of x, that is, $2^{[x+1]} \geq [x+1]^2$. These are the statements $S(x)$ and $S(x+1)$ in the induction principle; the fact that we are using x instead of n as the parameter should not be of concern; x or n is just a local variable.

As in Theorem 1.16, we should rewrite $S(x+1)$ so it can make use of $S(x)$. In this case, we can write $2^{[x+1]}$ as 2×2^x. Since $S(x)$ tells us that $2^x \geq x^2$, we can conclude that $2^{x+1} = 2 \times 2^x \geq 2x^2$.

But we need something different; we need to show that $2^{x+1} \geq (x+1)^2$. One way to prove this statement is to prove that $2x^2 \geq (x+1)^2$ and then use the transitivity of $\geq$ to show $2^{x+1} \geq 2x^2 \geq (x+1)^2$. In our proof that

$$2x^2 \geq (x+1)^2 \tag{1.7}$$

we may use the assumption that $x \geq 4$. Begin by simplifying (1.7):

$$x^2 \geq 2x + 1 \tag{1.8}$$

Divide (1.8) by x, to get:

Integers as Recursively Defined Concepts

We mentioned that inductive proofs are useful when the subject matter is recursively defined. However, our first examples were inductions on integers, which we do not normally think of as "recursively defined." However, there is a natural, recursive definition of when a number is a nonnegative integer, and this definition does indeed match the way inductions on integers proceed: from objects defined first, to those defined later.

BASIS: 0 is an integer.

INDUCTION: If n is an integer, then so is $n + 1$.

$$x \geq 2 + \frac{1}{x} \tag{1.9}$$

Since $x \geq 4$, we know $1/x \leq 1/4$. Thus, the left side of (1.9) is at least 4, and the right side is at most 2.25. We have thus proved the truth of (1.9). Therefore, Equations (1.8) and (1.7) are also true. Equation (1.7) in turn gives us $2x^2 \geq (x + 1)^2$ for $x \geq 4$ and lets us prove statement $S(x + 1)$, which we recall was $2^{x+1} \geq (x + 1)^2$. $\square$

1.4.2 More General Forms of Integer Inductions

Sometimes an inductive proof is made possible only by using a more general scheme than the one proposed in Section 1.4.1, where we proved a statement S for one basis value and then proved that "if $S(n)$ then $S(n+1)$." Two important generalizations of this scheme are:

1. We can use several basis cases. That is, we prove $S(i), S(i + 1), \ldots, S(j)$ for some $j > i$.

2. In proving $S(n + 1)$, we can use the truth of all the statements

$$S(i), S(i + 1), \ldots, S(n)$$

   rather than just using $S(n)$. Moreover, if we have proved basis cases up to $S(j)$, then we can assume $n \geq j$, rather than just $n \geq i$.

The conclusion to be made from this basis and inductive step is that $S(n)$ is true for all $n \geq i$.

Example 1.18: The following example will illustrate the potential of both principles. The statement $S(n)$ we would like to prove is that if $n \geq 8$, then n can be written as a sum of 3's and 5's. Notice, incidentally, that 7 cannot be written as a sum of 3's and 5's.

BASIS: The basis cases are $S(8)$, $S(9)$, and $S(10)$. The proofs are $8 = 3 + 5$, $9 = 3 + 3 + 3$, and $10 = 5 + 5$, respectively.

INDUCTION: Assume that $n \geq 10$ and that $S(8), S(9), \ldots, S(n)$ are true. We must prove $S(n+1)$ from these given facts. Our strategy is to subtract 3 from $n + 1$, observe that this number must be writable as a sum of 3's and 5's, and add one more 3 to the sum to get a way to write $n + 1$.

More formally, observe that $n - 2 \geq 8$, so we may assume $S(n - 2)$. That is, $n - 2 = 3a + 5b$ for some integers a and b. Then $n + 1 = 3 + 3a + 5b$, so $n + 1$ can be written as the sum of $a + 1$ 3's and b 5's. That proves $S(n + 1)$ and concludes the inductive step. $\square$

1.4.3 Structural Inductions

In automata theory, there are several recursively defined structures about which we need to prove statements. The familiar notions of trees and expressions are important examples. Like inductions, all recursive definitions have a basis case, where one or more elementary structures are defined, and an inductive step, where more complex structures are defined in terms of previously defined structures.

Example 1.19: Here is the recursive definition of a tree:

BASIS: A single node is a tree, and that node is the *root* of the tree.

INDUCTION: If $T_1, T_2, \ldots, T_k$ are trees, then we can form a new tree as follows:

1. Begin with a new node N, which is the root of the tree.

2. Add copies of all the trees $T_1, T_2, \ldots, T_k$.

3. Add edges from node N to the roots of each of the trees $T_1, T_2, \ldots, T_k$.

Figure 1.7 shows the inductive construction of a tree with root N from k smaller trees. $\square$

Example 1.20: Here is another recursive definition. This time we define *expressions* using the arithmetic operators $+$ and $*$, with both numbers and variables allowed as operands.

BASIS: Any number or letter (i.e., a variable) is an expression.

INDUCTION: If E and F are expressions, then so are $E + F$, $E * F$, and (E).

For example, both 2 and x are expressions by the basis. The inductive step tells us $x + 2$, $(x + 2)$, and $2 * (x + 2)$ are all expressions. Notice how each of these expressions depends on the previous ones being expressions. $\square$

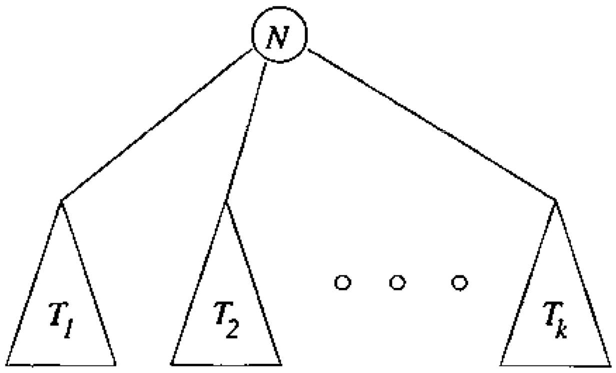


Figure 1.7: Inductive construction of a tree

Intuition Behind Structural Induction

We can suggest informally why structural induction is a valid proof method. Imagine the recursive definition establishing, one at a time, that certain structures $X_1, X_2, \ldots$ meet the definition. The basis elements come first, and the fact that X_i is in the defined set of structures can only depend on the membership in the defined set of structures that precede X_i on the list. Viewed this way, a structural induction is nothing but an induction on integer n of the statement $S(X_n)$. This induction may be of the generalized form discussed in Section 1.4.2, with multiple basis cases and an inductive step that uses all previous instances of the statement. However, we should remember, as explained in Section 1.4.1, that this intuition is not a formal proof, and in fact we must assume the validity this induction principle as we did the validity of the original induction principle of that section.

When we have a recursive definition, we can prove theorems about it using the following proof form, which is called ***structural induction***. Let $S(X)$ be a statement about the structures X that are defined by some particular recursive definition.

1. As a basis, prove $S(X)$ for the basis structure(s) X.

2. For the inductive step, take a structure X that the recursive definition says is formed from $Y_1, Y_2, \ldots, Y_k$. Assume that the statements $S(Y_1), S(Y_2), \ldots, S(Y_k)$, and use these to prove $S(X)$.

Our conclusion is that $S(X)$ is true for all X. The next two theorems are examples of facts that can be proved about trees and expressions.

Theorem 1.21 : Every tree has one more node than it has edges.

PROOF: The formal statement $S(T)$ we need to prove by structural induction is: "if T is a tree, and T has n nodes and e edges, then $n = e + 1$."

BASIS: The basis case is when T is a single node. Then $n = 1$ and $e = 0$, so the relationship $n = e + 1$ holds.

INDUCTION: Let T be a tree built by the inductive step of the definition, from root node N and k smaller trees $T_1, T_2, \ldots, T_k$. We may assume that the statements $S(T_i)$ hold for $i = 1, 2, \ldots, k$. That is, let T_i have n_i nodes and e_i edges; then $n_i = e_i + 1$.

The nodes of T are node N and all the nodes of the T_i's. There are thus $1 + n_1 + n_2 + \cdots + n_k$ nodes in T. The edges of T are the k edges we added explicitly in the inductive definition step, plus the edges of the T_i's. Hence, T has

$$k + e_1 + e_2 + \cdots + e_k \tag{1.10}$$

edges. If we substitute $e_i + 1$ for n_i in the count of the number of nodes of T we find that T has

$$1 + [e_1 + 1] + [e_2 + 1] + \cdots + [e_k + 1] \tag{1.11}$$

nodes. Since there are k of the "+1" terms in (1.10), we can regroup (1.11) as

$$k + 1 + e_1 + e_2 + \cdots + e_k \tag{1.12}$$

This expression is exactly 1 more than the expression of (1.10) that was given for the number of edges of T. Thus, T has one more node than it has edges. □

Theorem 1.22 : Every expression has an equal number of left and right parentheses.

PROOF: Formally, we prove the statement $S(G)$ about any expression G that is defined by the recursion of Example 1.20: the numbers of left and right parentheses in G are the same.

BASIS: If G is defined by the basis, then G is a number or variable. These expressions have 0 left parentheses and 0 right parentheses, so the numbers are equal.

INDUCTION: There are three rules whereby expression G may have been constructed according to the inductive step in the definition:

1. $G = E + F$.

2. $G = E * F$.

3. $G = (E)$.

We may assume that $S(E)$ and $S(F)$ are true; that is, E has the same number of left and right parentheses, say n of each, and F likewise has the same number of left and right parentheses, say m of each. Then we can compute the numbers of left and right parentheses in G for each of the three cases, as follows:

1. If $G = E + F$, then G has $n + m$ left parentheses and $n + m$ right parentheses; n of each come from E and m of each come from F.

2. If $G = E * F$, the count of parentheses for G is again $n + m$ of each, for the same reason as in case (1).

3. If $G = (E)$, then there are $n+1$ left parentheses in G — one left parenthesis is explicitly shown, and the other n are present in E. Likewise, there are $n + 1$ right parentheses in G; one is explicit and the other n are in E.

In each of the three cases, we see that the numbers of left and right parentheses in G are the same. This observation completes the inductive step and completes the proof. $\square$

1.4.4 Mutual Inductions

Sometimes, we cannot prove a single statement by induction, but rather need to prove a group of statements $S_1(n), S_2(n), \ldots, S_k(n)$ together by induction on n. Automata theory provides many such situations. In Example 1.23 we sample the common situation where we need to explain what an automaton does by proving a group of statements, one for each state. These statements tell under what sequences of inputs the automaton gets into each of the states.

Strictly speaking, proving a group of statements is no different from proving the *conjunction* (logical AND) of all the statements. For instance, the group of statements $S_1(n), S_2(n), \ldots, S_k(n)$ could be replaced by the single statement $S_1(n)$ AND $S_2(n)$ AND $\cdots$ AND $S_k(n)$. However, when there are really several independent statements to prove, it is generally less confusing to keep the statements separate and to prove them all in their own parts of the basis and inductive steps. We call this sort of proof *mutual induction*. An example will illustrate the necessary steps for a mutual recursion.

Example 1.23: Let us revisit the on/off switch, which we represented as an automaton in Example 1.1. The automaton itself is reproduced as Fig. 1.8. Since pushing the button switches the state between *on* and *off*, and the switch starts out in the *off* state, we expect that the following statements will together explain the operation of the switch:

$S_1(n)$: The automaton is in state *off* after n pushes if and only if n is even.

$S_2(n)$: The automaton is in state *on* after n pushes if and only if n is odd.

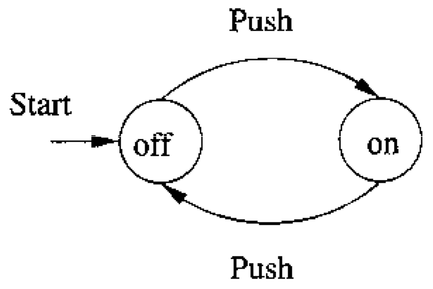


Figure 1.8: Repeat of the automaton of Fig. 1.1

We might suppose that S_1 implies S_2 and vice-versa, since we know that a number n cannot be both even and odd. However, what is not always true about an automaton is that it is in one and only one state. It happens that the automaton of Fig. 1.8 is always in exactly one state, but that fact must be proved as part of the mutual induction.

We give the basis and inductive parts of the proofs of statements $S_1(n)$ and $S_2(n)$ below. The proofs depend on several facts about odd and even integers: if we add or subtract 1 from an even integer, we get an odd integer, and if we add or subtract 1 from an odd integer we get an even integer.

BASIS: For the basis, we choose $n = 0$. Since there are two statements, each of which must be proved in both directions (because S_1 and S_2 are each "if-and-only-if" statements), there are actually four cases to the basis, and four cases to the induction as well.

1. [S_1; If] Since 0 is in fact even, we must show that after 0 pushes, the automaton of Fig. 1.8 is in state *off*. Since that is the start state, the automaton is indeed in state *off* after 0 pushes.

2. [S_1; Only-if] The automaton is in state *off* after 0 pushes, so we must show that 0 is even. But 0 is even by definition of "even," so there is nothing more to prove.

3. [S_2; If] The hypothesis of the "if" part of S_2 is that 0 is odd. Since this hypothesis H is false, any statement of the form "if H then C" is true, as we discussed in Section 1.3.2. Thus, this part of the basis also holds.

4. [S_2; Only-if] The hypothesis, that the automaton is in state *on* after 0 pushes, is also false, since the only way to get to state *on* is by following an arc labeled *Push*, which requires that the button be pushed at least once. Since the hypothesis is false, we can again conclude that the if-then statement is true.

INDUCTION: Now, we assume that $S_1(n)$ and $S_2(n)$ are true, and try to prove $S_1(n+1)$ and $S_2(n+1)$. Again, the proof separates into four parts.

1. [$S_1(n+1)$; If] The hypothesis for this part is that $n+1$ is even. Thus, n is odd. The "if" part of statement $S_2(n)$ says that after n pushes, the automaton is in state *on*. The arc from *on* to *off* labeled *Push* tells us that the $(n+1)$st push will cause the automaton to enter state *off*. That completes the proof of the "if" part of $S_1(n+1)$.

2. [$S_1(n+1)$; Only-if] The hypothesis is that the automaton is in state *off* after $n+1$ pushes. Inspecting the automaton of Fig. 1.8 tells us that the only way to get to state *off* after one or more moves is to be in state *on* and receive an input *Push*. Thus, if we are in state *off* after $n+1$ pushes, we must have been in state *on* after n pushes. Then, we may use the "only-if" part of statement $S_2(n)$ to conclude that n is odd. Consequently, $n+1$ is even, which is the desired conclusion for the only-if portion of $S_1(n+1)$.

3. [$S_2(n+1)$; If] This part is essentially the same as part (1), with the roles of statements S_1 and S_2 exchanged, and with the roles of "odd" and "even" exchanged. The reader should be able to construct this part of the proof easily.

4. [$S_2(n+1)$; Only-if] This part is essentially the same as part (2), with the roles of statements S_1 and S_2 exchanged, and with the roles of "odd" and "even" exchanged.

□

We can abstract from Example 1.23 the pattern for all mutual inductions:

- Each of the statements must be proved separately in the basis and in the inductive step.

- If the statements are "if-and-only-if," then both directions of each statement must be proved, both in the basis and in the induction.

1.5 The Central Concepts of Automata Theory

In this section we shall introduce the most important definitions of terms that pervade the theory of automata. These concepts include the "alphabet" (a set of symbols), "strings" (a list of symbols from an alphabet), and "language" (a set of strings from the same alphabet).

1.5.1 Alphabets

An *alphabet* is a finite, nonempty set of symbols. Conventionally, we use the symbol Σ for an alphabet. Common alphabets include:

1. $\Sigma = \{0, 1\}$, the *binary* alphabet.

2. $\Sigma = \{a, b, \ldots, z\}$, the set of all lower-case letters.

3. The set of all ASCII characters, or the set of all printable ASCII characters.

1.5.2 Strings

A *string* (or sometimes *word*) is a finite sequence of symbols chosen from some alphabet. For example, 01101 is a string from the binary alphabet $\Sigma = \{0, 1\}$. The string 111 is another string chosen from this alphabet.

The Empty String

The *empty string* is the string with zero occurrences of symbols. This string, denoted ϵ, is a string that may be chosen from any alphabet whatsoever.

Length of a String

It is often useful to classify strings by their *length*, that is, the number of positions for symbols in the string. For instance, 01101 has length 5. It is common to say that the length of a string is "the number of symbols" in the string; this statement is colloquially accepted but not strictly correct. Thus, there are only two symbols, 0 and 1, in the string 01101, but there are five *positions* for symbols, and its length is 5. However, you should generally expect that "the number of symbols" can be used when "number of positions" is meant.

The standard notation for the length of a string w is $|w|$. For example, $|011| = 3$ and $|\epsilon| = 0$.

Powers of an Alphabet

If Σ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define Σ^k to be the set of strings of length k, each of whose symbols is in Σ.

Example 1.24: Note that $\Sigma^0 = \{\epsilon\}$, regardless of what alphabet Σ is. That is, ϵ is the only string whose length is 0.

If $\Sigma = \{0, 1\}$, then $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$,

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

and so on. Note that there is a slight confusion between Σ and Σ^1. The former is an alphabet; its members 0 and 1 are symbols. The latter is a set of strings; its members are the strings 0 and 1, each of which is of length 1. We shall not try to use separate notations for the two sets, relying on context to make it clear whether $\{0, 1\}$ or similar sets are alphabets or sets of strings. $\square$

Type Convention for Symbols and Strings

Commonly, we shall use lower-case letters at the beginning of the alphabet (or digits) to denote symbols, and lower-case letters near the end of the alphabet, typically w, x, y, and z, to denote strings. You should try to get used to this convention, to help remind you of the types of the elements being discussed.

The set of all strings over an alphabet Σ is conventionally denoted Σ^*. For instance, $\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Put another way,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

Sometimes, we wish to exclude the empty string from the set of strings. The set of nonempty strings from alphabet Σ is denoted Σ^+. Thus, two appropriate equivalences are:

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$.
- $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

Concatenation of Strings

Let x and y be strings. Then xy denotes the *concatenation* of x and y, that is, the string formed by making a copy of x and following it by a copy of y. More precisely, if x is the string composed of i symbols $x = a_1a_2 \cdots a_i$ and y is the string composed of j symbols $y = b_1b_2 \cdots b_j$, then xy is the string of length $i + j$: $xy = a_1a_2 \cdots a_ib_1b_2 \cdots b_j$.

Example 1.25: Let $x = 01101$ and $y = 110$. Then $xy = 01101110$ and $yx = 11001101$. For any string w, the equations $\epsilon w = w\epsilon = w$ hold. That is, ϵ is the *identity for concatenation*, since when concatenated with any string it yields the other string as a result (analogously to the way 0, the identity for addition, can be added to any number x and yields x as a result). □

1.5.3 Languages

A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet, is called a *language*. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a *language over* Σ. Notice that a language over Σ need not include strings with all the symbols of Σ, so once we have established that L is a language over Σ, we also know it is a language over any alphabet that is a superset of Σ.

The choice of the term "language" may seem strange. However, common languages can be viewed as sets of strings. An example is English, where the collection of legal English words is a set of strings over the alphabet that consists of all the letters. Another example is C, or any other programming language, where the legal programs are a subset of the possible strings that can be formed from the alphabet of the language. This alphabet is a subset of the ASCII characters. The exact alphabet may differ slightly among different programming languages, but generally includes the upper- and lower-case letters, the digits, punctuation, and mathematical symbols.

However, there are also many other languages that appear when we study automata. Some are abstract examples, such as:

1. The language of all strings consisting of n 0's followed by n 1's, for some $n \geq 0$: $\{\epsilon, 01, 0011, 000111, \ldots\}$.

2. The set of strings of 0's and 1's with an equal number of each:

$$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

3. The set of binary numbers whose value is a prime:

$$\{10, 11, 101, 111, 1011, \ldots\}$$

4. Σ^* is a language for any alphabet Σ.

5. $\emptyset$, the empty language, is a language over any alphabet.

6. $\{\epsilon\}$, the language consisting of only the empty string, is also a language over any alphabet. Notice that $\emptyset \neq \{\epsilon\}$; the former has no strings and the latter has one string.

The only important constraint on what can be a language is that all alphabets are finite. Thus languages, although they can have an infinite number of strings, are restricted to consist of strings drawn from one fixed, finite alphabet.

1.5.4 Problems

In automata theory, a ***problem*** is the question of deciding whether a given string is a member of some particular language. It turns out, as we shall see, that anything we more colloquially call a "problem" can be expressed as membership in a language. More precisely, if Σ is an alphabet, and L is a language over Σ, then the problem L is:

- Given a string w in Σ^*, decide whether or not w is in L.

Example 1.26: The problem of testing primality can be expressed by the language L_p consisting of all binary strings whose value as a binary number is a prime. That is, given a string of 0's and 1's, say "yes" if the string is the binary representation of a prime and say "no" if not. For some strings, this

Set-Formers as a Way to Define Languages

It is common to describe a language using a "set-former":

$$\{w \mid \text{something about } w\}$$

This expression is read "the set of words w such that (whatever is said about w to the right of the vertical bar)." Examples are:

1. $\{w \mid w$ consists of an equal number of 0's and 1's $\}$.
2. $\{w \mid w$ is a binary integer that is prime $\}$.
3. $\{w \mid w$ is a syntactically correct C program $\}$.

It is also common to replace w by some expression with parameters and describe the strings in the language by stating conditions on the parameters. Here are some examples; the first with parameter n, the second with parameters i and j:

1. $\{0^n1^n \mid n \geq 1\}$. Read "the set of 0 to the n 1 to the n such that n is greater than or equal to 1," this language consists of the strings $\{01, 0011, 000111, \ldots\}$. Notice that, as with alphabets, we can raise a single symbol to a power n in order to represent n copies of that symbol.
2. $\{0^i1^j \mid 0 \leq i \leq j\}$. This language consists of strings with some 0's (possibly none) followed by at least as many 1's.

decision is easy. For instance, 0011101 cannot be the representation of a prime, for the simple reason that every integer except 0 has a binary representation that begins with 1. However, it is less obvious whether the string 11101 belongs to L_p, so any solution to this problem will have to use significant computational resources of some kind: time and/or space, for example. □

One potentially unsatisfactory aspect of our definition of "problem" is that one commonly thinks of problems not as decision questions (is or is not the following true?) but as requests to compute or transform some input (find the best way to do this task). For instance, the task of the parser in a C compiler can be thought of as a problem in our formal sense, where one is given an ASCII string and asked to decide whether or not the string is a member of L_c, the set of valid C programs. However, the parser does more than decide. It produces a parse tree, entries in a symbol table and perhaps more. Worse, the compiler as a whole solves the problem of turning a C program into object code for some

Is It a Language or a Problem?

Languages and problems are really the same thing. Which term we prefer to use depends on our point of view. When we care only about strings for their own sake, e.g., in the set $\{0^n1^n \mid n \geq 1\}$, then we tend to think of the set of strings as a language. In the last chapters of this book, we shall tend to assign "semantics" to the strings, e.g., think of strings as coding graphs, logical expressions, or even integers. In those cases, where we care more about the thing represented by the string than the string itself, we shall tend to think of a set of strings as a problem.

machine, which is far from simply answering "yes" or "no" about the validity of a program.

Nevertheless, the definition of "problems" as languages has stood the test of time as the appropriate way to deal with the important questions of complexity theory. In this theory, we are interested in proving lower bounds on the complexity of certain problems. Especially important are techniques for proving that certain problems cannot be solved in an amount of time that is less than exponential in the size of their input. It turns out that the yes/no or language-based version of known problems are just as hard in this sense, as their "solve this" versions.

That is, if we can prove it is hard to decide whether a given string belongs to the language L_X of valid strings in programming language X, then it stands to reason that it will not be easier to translate programs in language X to object code. For if it were easy to generate code, then we could run the translator, and conclude that the input was a valid member of L_X exactly when the translator succeeded in producing object code. Since the final step of determining whether object code has been produced cannot be hard, we can use the fast algorithm for generating the object code to decide membership in L_X efficiently. We thus contradict the assumption that testing membership in L_X is hard. We have a proof by contradiction of the statement "if testing membership in L_X is hard, then compiling programs in programming language X is hard."

This technique, showing one problem hard by using its supposed efficient algorithm to solve efficiently another problem that is already known to be hard, is called a "reduction" of the second problem to the first. It is an essential tool in the study of the complexity of problems, and it is facilitated greatly by our notion that problems are questions about membership in a language, rather than more general kinds of questions.

1.6 Summary of Chapter 1

✦ *Finite Automata*: Finite automata involve states and transitions among states in response to inputs. They are useful for building several different kinds of software, including the lexical analysis component of a compiler and systems for verifying the correctness of circuits or protocols, for example.

✦ *Regular Expressions*: These are a structural notation for describing the same patterns that can be represented by finite automata. They are used in many common types of software, including tools to search for patterns in text or in file names, for instance.

✦ *Context-Free Grammars*: These are an important notation for describing the structure of programming languages and related sets of strings; they are used to build the parser component of a compiler.

✦ *Turing Machines*: These are automata that model the power of real computers. They allow us to study decidabilty, the question of what can or cannot be done by a computer. They also let us distinguish tractable problems — those that can be solved in polynomial time — from the intractable problems — those that cannot.

✦ *Deductive Proofs*: This basic method of proof proceeds by listing statements that are either given to be true, or that follow logically from some of the previous statements.

✦ *Proving If-Then Statements*: Many theorems are of the form "if (something) then (something else)." The statement or statements following the "if" are the hypothesis, and what follows "then" is the conclusion. Deductive proofs of if-then statements begin with the hypothesis, and continue with statements that follow logically from the hypothesis and previous statements, until the conclusion is proved as one of the statements.

✦ *Proving If-And-Only-If Statements*: There are other theorems of the form "(something) if and only if (something else)." They are proved by showing if-then statements in both directions. A similar kind of theorem claims the equality of the sets described in two different ways; these are proved by showing that each of the two sets is contained in the other.

✦ *Proving the Contrapositive*: Sometimes, it is easier to prove a statement of the form "if H then C" by proving the equivalent statement: "if not C then not H." The latter is called the contrapositive of the former.

✦ *Proof by Contradiction*: Other times, it is more convenient to prove the statement "if H then C" by proving "if H and not C then (something known to be false)." A proof of this type is called proof by contradiction.

✦ *Counterexamples*: Sometimes we are asked to show that a certain statement is not true. If the statement has one or more parameters, then we can show it is false as a generality by providing just one counterexample, that is, one assignment of values to the parameters that makes the statement false.

✦ *Inductive Proofs*: A statement that has an integer parameter n can often by proved by induction on n. We prove the statement is true for the basis, a finite number of cases for particular values of n, and then prove the inductive step: that if the statement is true for values up to n, then it is true for $n + 1$.

✦ *Structural Inductions*: In some situations, including many in this book, the theorem to be proved inductively is about some recursively defined construct, such as trees. We may prove a theorem about the constructed objects by induction on the number of steps used in its construction. This type of induction is referred to as structural.

✦ *Alphabets*: An alphabet is any finite set of symbols.

✦ *Strings*: A string is a finite-length sequence of symbols.

✦ *Languages and Problems*: A language is a (possibly infinite) set of strings, all of which choose their symbols from some one alphabet. When the strings of a language are to be interpreted in some way, the question of whether a string is in the language is sometimes called a problem.

1.7 References for Chapter 1

For extended coverage of the material of this chapter, including mathematical concepts underlying Computer Science, we recommend [1].

1. A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1994.

Chapter 2

Finite Automata

This chapter introduces the class of languages known as "regular languages." These languages are exactly the ones that can be described by finite automata, which we sampled briefly in Section 1.1.1. After an extended example that will provide motivation for the study to follow, we define finite automata formally.

As was mentioned earlier, a finite automaton has a set of states, and its "control" moves from state to state in response to external "inputs." One of the crucial distinctions among classes of finite automata is whether that control is "deterministic," meaning that the automaton cannot be in more than one state at any one time, or "nondeterministic," meaning that it may be in several states at once. We shall discover that adding nondeterminism does not let us define any language that cannot be defined by a deterministic finite automaton, but there can be substantial efficiency in describing an application using a nondeterministic automaton. In effect, nondeterminism allows us to "program" solutions to problems using a higher-level language. The nondeterministic finite automaton is then "compiled," by an algorithm we shall learn in this chapter, into a deterministic automaton that can be "executed" on a conventional computer.

We conclude the chapter with a study of an extended nondeterministic automaton that has the additional choice of making a transition from one state to another spontaneously, i.e., on the empty string as "input." These automata also accept nothing but the regular languages. However, we shall find them quite important in Chapter 3, when we study regular expressions and their equivalence to automata.

The study of the regular languages continues in Chapter 3. There, we introduce another important way to describe regular languages: the algebraic notation known as regular expressions. After discussing regular expressions, and showing their equivalence to finite automata, we use both automata and regular expressions as tools in Chapter 4 to show certain important properties of the regular languages. Examples of such properties are the "closure" properties, which allow us to claim that one language is regular because one or more other languages are known to be regular, and "decision" properties. The latter are algorithms to answer questions about automata or regular expressions, e.g., whether two automata or expressions represent the same language.

2.1 An Informal Picture of Finite Automata

In this section, we shall study an extended example of a real-world problem whose solution uses finite automata in an important role. We investigate protocols that support "electronic money" — files that a customer can use to pay for goods on the internet, and that the seller can receive with assurance that the "money" is real. The seller must know that the file has not been forged, nor has it been copied and sent to the seller, while the customer retains a copy of the same file to spend again.

The nonforgeability of the file is something that must be assured by a bank and by a cryptography policy. That is, a third player, the bank, must issue and encrypt the "money" files, so that forgery is not a problem. However, the bank has a second important job: it must keep a database of all the valid money that it has issued, so that it can verify to a store that the file it has received represents real money and can be credited to the store's account. We shall not address the cryptographic aspects of the problem, nor shall we worry about how the bank can store and retrieve what could be billions of "electronic dollar bills." These problems are not likely to represent long-term impediments to the concept of electronic money, and examples of its small-scale use have existed since the late 1990's.

However, in order to use electronic money, protocols need to be devised to allow the manipulation of the money in a variety of ways that the users want. Because monetary systems always invite fraud, we must verify whatever policy we adopt regarding how money is used. That is, we need to prove the only things that can happen are things we intend to happen — things that do not allow an unscrupulous user to steal from others or to "manufacture" money. In the balance of this section, we shall introduce a very simple example of a (bad) electronic-money protocol, model it with finite automata, and show how constructions on automata can be used to verify protocols (or, in this case, to discover that the protocol has a bug).

2.1.1 The Ground Rules

There are three participants: the customer, the store, and the bank. We assume for simplicity that there is only one "money" file in existence. The customer may decide to transfer this money file to the store, which will then redeem the file from the bank (i.e., get the bank to issue a new money file belonging to the store rather than the customer) and ship goods to the customer. In addition, the customer has the option to cancel the file. That is, the customer may ask the bank to place the money back in the customer's account, making the money no longer spendable. Interaction among the three participants is thus limited to five events:

1. The customer may decide to *pay*. That is, the customer sends the money to the store.
2. The customer may decide to *cancel*. The money is sent to the bank with a message that the value of the money is to be added to the customer's bank account.
3. The store may *ship* goods to the customer.
4. The store may *redeem* the money. That is, the money is sent to the bank with a request that its value be given to the store.
5. The bank may *transfer* the money by creating a new, suitably encrypted money file and sending it to the store.

2.1.2 The Protocol

The three participants must design their behaviors carefully, or the wrong things may happen. In our example, we make the reasonable assumption that the customer cannot be relied upon to act responsibly. In particular, the customer may try to copy the money file, use it to pay several times, or both pay and cancel the money, thus getting the goods "for free."

The bank must behave responsibly, or it cannot be a bank. In particular, it must make sure that two stores cannot both redeem the same money file, and it must not allow money to be both canceled and redeemed. The store should be careful as well. In particular, it should not ship goods until it is sure it has been given valid money for the goods.

Protocols of this type can be represented as finite automata. Each state represents a situation that one of the participants could be in. That is, the state "remembers" that certain important events have happened and that others have not yet happened. Transitions between states occur when one of the five events described above occur. We shall think of these events as "external" to the automata representing the three participants, even though each participant is responsible for initiating one or more of the events. It turns out that what is important about the problem is what sequences of events can happen, not who is allowed to initiate them.

Figure 2.1 represents the three participants by automata. In that diagram, we show only the events that affect a participant. For example, the action *pay* affects only the customer and store. The bank does not know that the money has been sent by the customer to the store; it discovers that fact only when the store executes the action *redeem*.

Let us examine first the automaton (c) for the bank. The start state is state 1; it represents the situation where the bank has issued the money file in question but has not been requested either to redeem it or to cancel it. If a

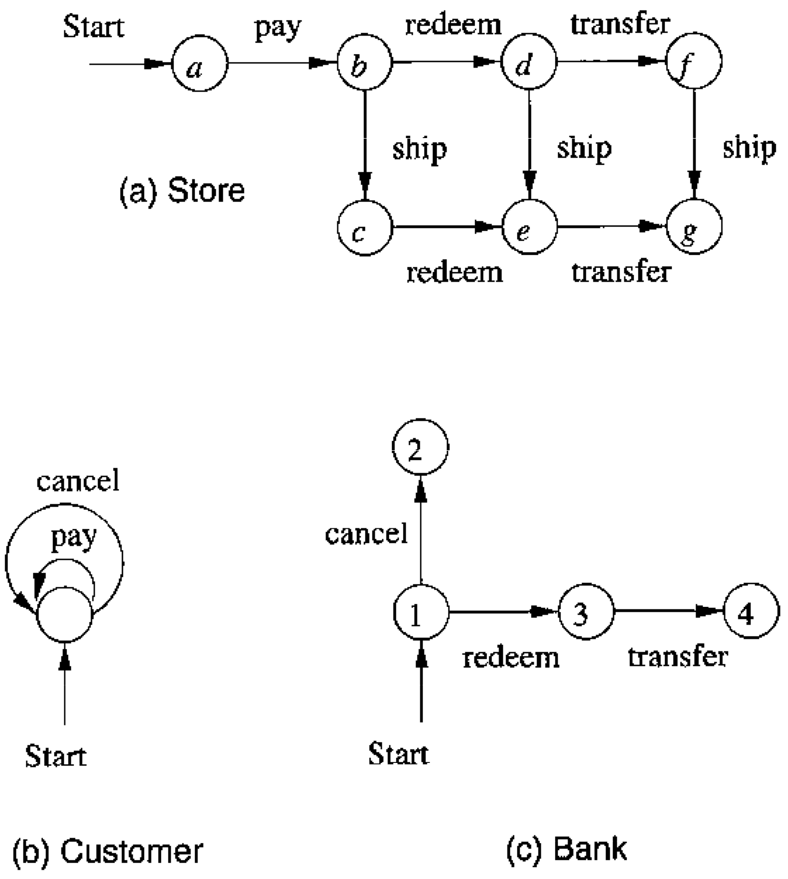


Figure 2.1: Finite automata representing a customer, a store, and a bank

cancel request is sent to the bank by the customer, then the bank restores the money to the customer's account and enters state 2. The latter state represents the situation where the money has been cancelled. The bank, being responsible, will not leave state 2 once it is entered, since the bank must not allow the same money to be cancelled again or spent by the customer.[1]

Alternatively, when in state 1 the bank may receive a *redeem* request from the store. If so, it goes to state 3, and shortly sends the store a *transfer* message, with a new money file that now belongs to the store. After sending the transfer message, the bank goes to state 4. In that state, it will neither accept *cancel* or *redeem* requests nor will it perform any other actions regarding this particular money file.

Now, let us consider Fig. 2.1(a), the automaton representing the actions of the store. While the bank always does the right thing, the store's system has some defects. Imagine that the shipping and financial operations are done by

[1]You should remember that this entire discussion is about one single money file. The bank will in fact be running the same protocol with a large number of electronic pieces of money, but the workings of the protocol are the same for each of them, so we can discuss the problem as if there were only one piece of electronic money in existence.

separate processes, so there is the opportunity for the *ship* action to be done either before, after, or during the redemption of the electronic money. That policy allows the store to get into a situation where it has already shipped the goods and then finds out the money was bogus.

The store starts out in state a. When the customer orders the goods by performing the *pay* action, the store enters state b. In this state, the store begins both the shipping and redemption processes. If the goods are shipped first, then the store enters state c, where it must still redeem the money from the bank and receive the *transfer* of an equivalent money file from the bank. Alternatively, the store may send the *redeem* message first, entering state d. From state d, the store might next ship, entering state e, or it might next receive the transfer of money from the bank, entering state f. From state f, we expect that the store will eventually ship, putting the store in state g, where the transaction is complete and nothing more will happen. In state e, the store is waiting for the *transfer* from the bank. Unfortunately, the goods have already been shipped, and if the *transfer* never occurs, the store is out of luck.

Last, observe the automaton for the customer, Fig. 2.1(b). This automaton has only one state, reflecting the fact that the customer "can do anything." The customer can perform the *pay* and *cancel* actions any number of times, in any order, and stays in the lone state after each action.

2.1.3 Enabling the Automata to Ignore Actions

While the three automata of Fig. 2.1 reflect the behaviors of the three participants independently, there are certain transitions that are missing. For example, the store is not affected by a *cancel* message, so if the *cancel* action is performed by the customer, the store should remain in whatever state it is in. However, in the formal definition of a finite automaton, which we shall study in Section 2.2, whenever an input X is received by an automaton, the automaton must follow an arc labeled X from the state it is in to some new state. Thus, the automaton for the store needs an additional arc from each state to itself, labeled *cancel*. Then, whenever the *cancel* action is executed, the store automaton can make a "transition" on that input, with the effect that it stays in the same state it was in. Without these additional arcs, whenever the *cancel* action was executed the store automaton would "die"; that is, the automaton would be in no state at all, and further actions by that automaton would be impossible.

Another potential problem is that one of the participants may, intentionally or erroneously, send an unexpected message, and we do not want this action to cause one of the automata to die. For instance, suppose the customer decided to execute the *pay* action a second time, while the store was in state e. Since that state has no arc out with label *pay*, the store's automaton would die before it could receive the transfer from the bank. In summary, we must add to the automata of Fig. 2.1 loops on certain states, with labels for all those actions that must be ignored when in that state; the complete automata are shown in Fig. 2.2. To save space, we combine the labels onto one arc, rather than showing several arcs with the same heads and tails but different labels. The two kinds of actions that must be ignored are:

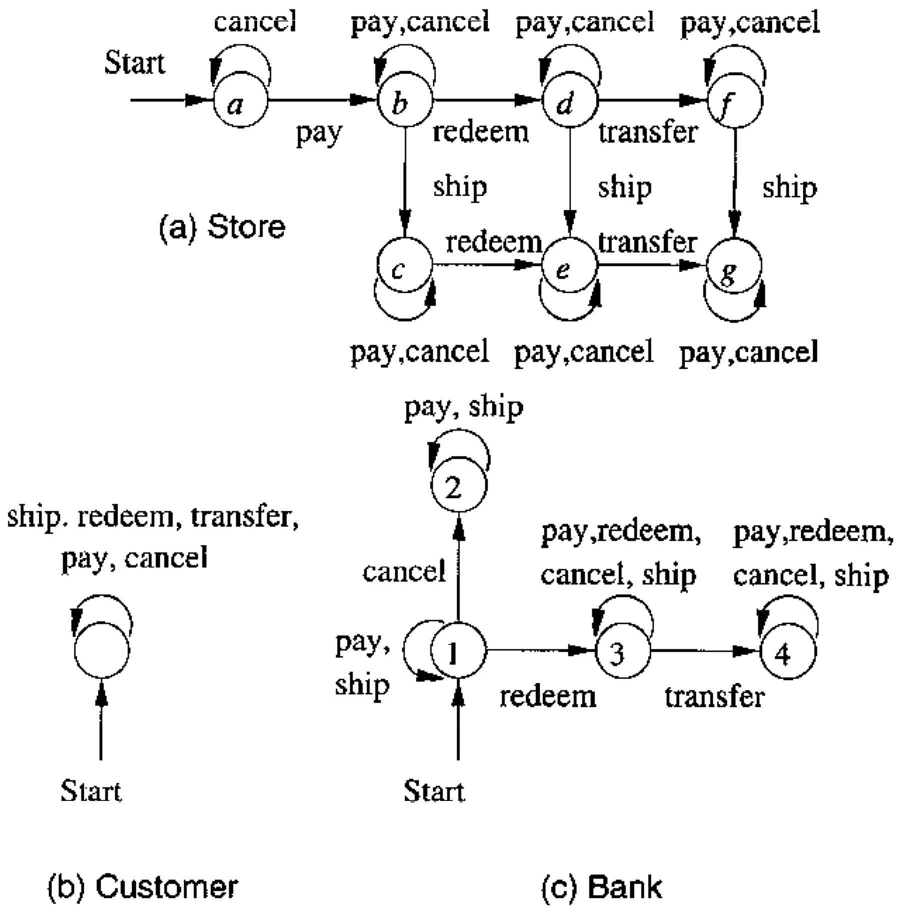


Figure 2.2: The complete sets of transitions for the three automata

1. *Actions that are irrelevant to the participant involved.* As we saw, the only irrelevant action for the store is *cancel*, so each of its seven states has a loop labeled *cancel*. For the bank, both *pay* and *ship* are irrelevant, so we have put at each of the bank's states an arc labeled *pay, ship*. For the customer, *ship*, *redeem* and *transfer* are all irrelevant, so we add arcs with these labels. In effect, it stays in its one state on any sequence of inputs, so the customer automaton has no effect on the operation of the overall system. Of course, the customer is still a participant, since it is the customer who initiates the *pay* and *cancel* actions. However, as we mentioned, the matter of who initiates actions has nothing to do with the behavior of the automata.

2. *Actions that must not be allowed to kill an automaton.* As mentioned, we must not allow the customer to kill the store's automaton by executing *pay*

again, so we have added loops with label *pay* to all but state a (where the *pay* action is expected and relevant). We have also added loops with labels *cancel* to states 3 and 4 of the bank, in order to prevent the customer from killing the bank's automaton by trying to cancel money that has already been redeemed. The bank properly ignores such a request. Likewise, states 3 and 4 have loops on *redeem*. The store should not try to redeem the same money twice, but if it does, the bank properly ignores the second request.

2.1.4 The Entire System as an Automaton

While we now have models for how the three participants behave, we do not yet have a representation for the interaction of the three participants. As mentioned, because the customer has no constraints on behavior, that automaton has only one state, and any sequence of events lets it stay in that state; i.e., it is not possible for the system as a whole to "die" because the customer automaton has no response to an action. However, both the store and bank behave in a complex way, and it is not immediately obvious in what combinations of states these two automata can be.

The normal way to explore the interaction of automata such as these is to construct the *product* automaton. That automaton's states represent a pair of states, one from the store and one from the bank. For instance, the state $(3, d)$ of the product automaton represents the situation where the bank is in state 3, and the store is in state d. Since the bank has four states and the store has seven, the product automaton has $4 \times 7 = 28$ states.

We show the product automaton in Fig. 2.3. For clarity, we have arranged the 28 states in an array. The row corresponds to the state of the bank and the column to the state of the store. To save space, we have also abbreviated the labels on the arcs, with P, S, C, R, and T standing for pay, ship, cancel, redeem, and transfer, respectively.

To construct the arcs of the product automaton, we need to run the bank and store automata "in parallel." Each of the two components of the product automaton independently makes transitions on the various inputs. However, it is important to notice that if an input action is received, and one of the two automata has no state to go to on that input, then the product automaton "dies"; it has no state to go to.

To make this rule for state transitions precise, suppose the product automaton is in state (i, x). That state corresponds to the situation where the bank is in state i and the store in state x. Let Z be one of the input actions. We look at the automaton for the bank, and see whether there is a transition out of state i with label Z. Suppose there is, and it leads to state j (which might be the same as i if the bank loops on input Z). Then, we look at the store and see if there is an arc labeled Z leading to some state y. If both j and y exist, then the product automaton has an arc from state (i, x) to state (j, y), labeled Z. If either of states j or y do not exist (because the bank or store has no arc

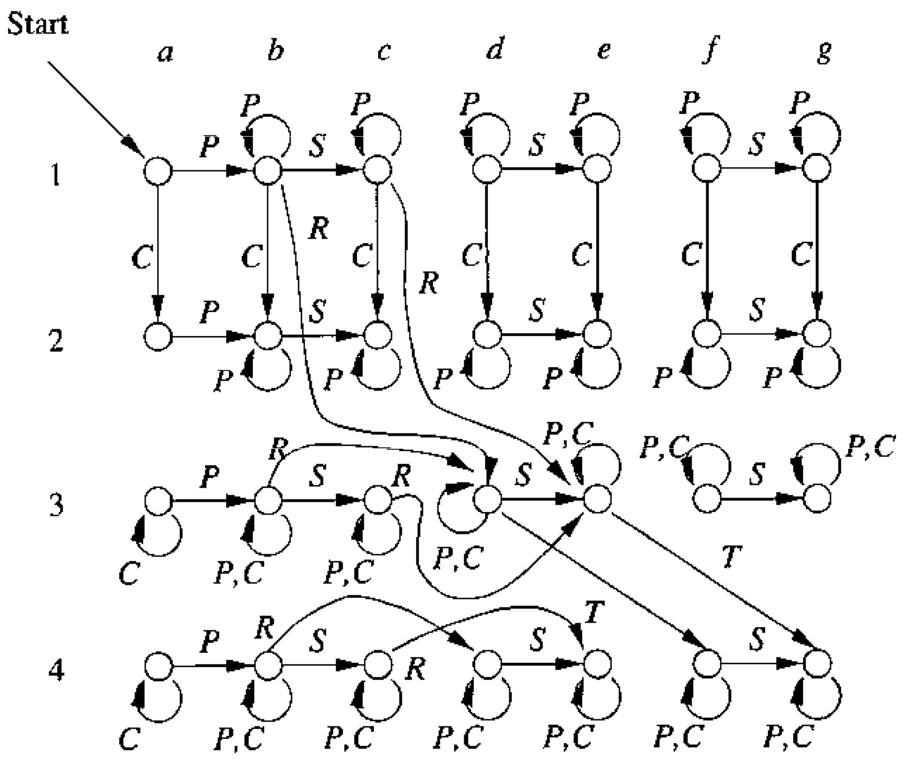


Figure 2.3: The product automaton for the store and bank

out of i or x, respectively, for input Z), then there is no arc out of (i, x) labeled Z.

We can now see how the arcs of Fig. 2.3 were selected. For instance, on input *pay*, the store goes from state a to b, but stays put if it is in any other state besides a. The bank stays in whatever state it is in when the input is *pay*, because that action is irrelevant to the bank. This observation explains the four arcs labeled P at the left ends of the four rows in Fig. 2.3, and the loops labeled P on other states.

For another example of how the arcs are selected, consider the input *redeem*. If the bank receives a *redeem* message when in state 1, it goes to state 3. If in states 3 or 4, it stays there, while in state 2 the bank automaton dies; i.e., it has nowhere to go. The store, on the other hand, can make transitions from state b to d or from c to e when the *redeem* input is received. In Fig. 2.3, we see six arcs labeled *redeem*, corresponding to the six combinations of three bank states and two store states that have outward-bound arcs labeled R. For example, in state $(1, b)$, the arc labeled R takes the automaton to state $(3, d)$, since *redeem* takes the bank from state 1 to 3 and the store from b to d. As another example, there is an arc labeled R from $(4, c)$ to $(4, e)$, since *redeem* takes the bank from state 4 back to state 4, while it takes the store from state c to state e.

2.1.5 Using the Product Automaton to Validate the Protocol

Figure 2.3 tells us some interesting things. For instance, of the 28 states, only ten of them can be reached from the start state, which is $(1, a)$ — the combination of the start states of the bank and store automata. Notice that states like $(2, e)$ and $(4, d)$ are not ***accessible***, that is, there is no path to them from the start state. Inaccessible states need not be included in the automaton, and we did so in this example just to be systematic.

However, the real purpose of analyzing a protocol such as this one using automata is to ask and answer questions that mean "can the following type of error occur?" In the example at hand, we might ask whether it is possible that the store can ship goods and never get paid. That is, can the product automaton get into a state in which the store has shipped (that is, the state is in column c, e, or g), and yet no transition on input T was ever made or will be made?

For instance, in state $(3, e)$, the goods have shipped, but there will eventually be a transition on input T to state $(4, g)$. In terms of what the bank is doing, once it has gotten to state 3, it has received the *redeem* request and processed it. That means it must have been in state 1 before receiving the *redeem* and therefore the *cancel* message had not been received and will be ignored if received in the future. Thus, the bank will eventually perform the transfer of money to the store.

However, state $(2, c)$ is a problem. The state is accessible, but the only arc out leads back to that state. This state corresponds to a situation where the bank received a *cancel* message before a *redeem* message. However, the store received a *pay* message; i.e., the customer was being duplicitous and has both spent and canceled the same money. The store foolishly shipped before trying to redeem the money, and when the store does execute the *redeem* action, the bank will not even acknowledge the message, because it is in state 2, where it has canceled the money and will not process a *redeem* request.

2.2 Deterministic Finite Automata

Now it is time to present the formal notion of a finite automaton, so that we may start to make precise some of the informal arguments and descriptions that we saw in Sections 1.1.1 and 2.1. We begin by introducing the formalism of a deterministic finite automaton, one that is in a single state after reading any sequence of inputs. The term "deterministic" refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state. In contrast, "nondeterministic" finite automata, the subject of Section 2.3, can be in several states at once. The term "finite automaton" will refer to the deterministic variety, although we shall use "deterministic" or the abbreviation *DFA* normally, to remind the reader of which kind of automaton we are talking about.

2.2.1 Definition of a Deterministic Finite Automaton

A *deterministic finite automaton* consists of:

1. A finite set of ***states***, often denoted Q.

2. A finite set of ***input symbols***, often denoted Σ.

3. A ***transition function*** that takes as arguments a state and an input symbol and returns a state. The transition function will commonly be denoted δ. In our informal graph representation of automata, δ was represented by arcs between states and the labels on the arcs. If q is a state, and a is an input symbol, then $\delta(q, a)$ is that state p such that there is an arc labeled a from q to p.[2]

4. A ***start state***, one of the states in Q.

5. A set of ***final*** or ***accepting*** states F. The set F is a subset of Q.

A deterministic finite automaton will often be referred to by its acronym: *DFA*. The most succinct representation of a DFA is a listing of the five components above. In proofs we often talk about a DFA in "five-tuple" notation:

$$A = (Q, \Sigma, \delta, q_0, F)$$

where A is the name of the DFA, Q is its set of states, Σ its input symbols, δ its transition function, q_0 its start state, and F its set of accepting states.

2.2.2 How a DFA Processes Strings

The first thing we need to understand about a DFA is how the DFA decides whether or not to "accept" a sequence of input symbols. The "language" of the DFA is the set of all strings that the DFA accepts. Suppose $a_1a_2\cdots a_n$ is a sequence of input symbols. We start out with the DFA in its start state, q_0. We consult the transition function δ, say $\delta(q_0, a_1) = q_1$ to find the state that the DFA A enters after processing the first input symbol a_1. We process the next input symbol, a_2, by evaluating $\delta(q_1, a_2)$; let us suppose this state is q_2. We continue in this manner, finding states $q_3, q_4, \ldots, q_n$ such that $\delta(q_{i-1}, a_i) = q_i$ for each i. If q_n is a member of F, then the input $a_1a_2\cdots a_n$ is accepted, and if not then it is "rejected."

Example 2.1: Let us formally specify a DFA that accepts all and only the strings of 0's and 1's that have the sequence 01 somewhere in the string. We can write this language L as:

$$\{w \mid w \text{ is of the form } x01y \text{ for some strings } x \text{ and } y \text{ consisting of 0's and 1's only}\}$$

[2]More accurately, the graph is a picture of some transition function δ, and the arcs of the graph are constructed to reflect the transitions specified by δ.

Another equivalent description, using parameters x and y to the left of the vertical bar, is:

$$\{x01y \mid x \text{ and } y \text{ are any strings of 0's and 1's}\}$$

Examples of strings in the language include 01, 11010, and 100011. Examples of strings *not* in the language include ϵ, 0, and 111000.

What do we know about an automaton that can accept this language L? First, its input alphabet is $\Sigma = \{0, 1\}$. It has some set of states, Q, of which one, say q_0, is the start state. This automaton has to remember the important facts about what inputs it has seen so far. To decide whether 01 is a substring of the input, A needs to remember:

1. Has it already seen 01? If so, then it accepts every sequence of further inputs; i.e., it will only be in accepting states from now on.

2. Has it never seen 01, but its most recent input was 0, so if it now sees a 1, it will have seen 01 and can accept everything it sees from here on?

3. Has it never seen 01, but its last input was either nonexistent (it just started) or it last saw a 1? In this case, A cannot accept until it first sees a 0 and then sees a 1 immediately after.

These three conditions can each be represented by a state. Condition (3) is represented by the start state, q_0. Surely, when just starting, we need to see a 0 and then a 1. But if in state q_0 we next see a 1, then we are no closer to seeing 01, and so we must stay in state q_0. That is, $\delta(q_0, 1) = q_0$.

However, if we are in state q_0 and we next see a 0, we are in condition (2). That is, we have never seen 01, but we have our 0. Thus, let us use q_2 to represent condition (2). Our transition from q_0 on input 0 is $\delta(q_0, 0) = q_2$.

Now, let us consider the transitions from state q_2. If we see a 0, we are no better off than we were, but no worse either. We have not seen 01, but 0 was the last symbol, so we are still waiting for a 1. State q_2 describes this situation perfectly, so we want $\delta(q_2, 0) = q_2$. If we are in state q_2 and we see a 1 input, we now know there is a 0 followed by a 1. We can go to an accepting state, which we shall call q_1, and which corresponds to condition (1) above. That is, $\delta(q_2, 1) = q_1$.

Finally, we must design the transitions for state q_1. In this state, we have already seen a 01 sequence, so regardless of what happens, we shall still be in a situation where we've seen 01. That is, $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.

Thus, $Q = \{q_0, q_1, q_2\}$. As we said, q_0 is the start state, and the only accepting state is q_1; that is, $F = \{q_1\}$. The complete specification of the automaton A that accepts the language L of strings that have a 01 substring, is

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

where δ is the transition function described above. $\square$

2.2.3 Simpler Notations for DFA's

Specifying a DFA as a five-tuple with a detailed description of the δ transition function is both tedious and hard to read. There are two preferred notations for describing automata:

1. A *transition diagram*, which is a graph such as the ones we saw in Section 2.1.

2. A *transition table*, which is a tabular listing of the δ function, which by implication tells us the set of states and the input alphabet.

Transition Diagrams

A *transition diagram* for a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

a) For each state in Q there is a node.

b) For each state q in Q and each input symbol a in Σ, let $\delta(q, a) = p$. Then the transition diagram has an arc from node q to node p, labeled a. If there are several input symbols that cause transitions from q to p, then the transition diagram can have one arc, labeled by the list of these symbols.

c) There is an arrow into the start state q_0, labeled *Start*. This arrow does not originate at any node.

d) Nodes corresponding to accepting states (those in F) are marked by a double circle. States not in F have a single circle.

Example 2.2: Figure 2.4 shows the transition diagram for the DFA that we designed in Example 2.1. We see in that diagram the three nodes that correspond to the three states. There is a *Start* arrow entering the start state, q_0, and the one accepting state, q_1, is represented by a double circle. Out of each state is one arc labeled 0 and one arc labeled 1 (although the two arcs are combined into one with a double label in the case of q_1). The arcs each correspond to one of the δ facts developed in Example 2.1. □

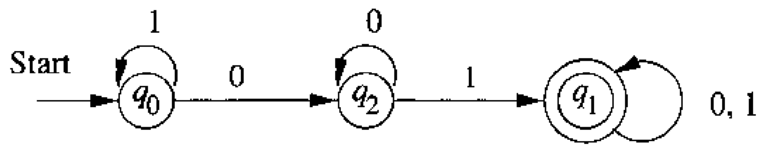


Figure 2.4: The transition diagram for the DFA accepting all strings with a substring 01

Transition Tables

A ***transition table*** is a conventional, tabular representation of a function like δ that takes two arguments and returns a value. The rows of the table correspond to the states, and the columns correspond to the inputs. The entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$.

Example 2.3: The transition table corresponding to the function δ of Example 2.1 is shown in Fig. 2.5. We have also shown two other features of a transition table. The start state is marked with an arrow, and the accepting states are marked with a star. Since we can deduce the sets of states and input symbols by looking at the row and column heads, we can now read from the transition table all the information we need to specify the finite automaton uniquely. □

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Figure 2.5: Transition table for the DFA of Example 2.1

2.2.4 Extending the Transition Function to Strings

We have explained informally that the DFA defines a language: the set of all strings that result in a sequence of state transitions from the start state to an accepting state. In terms of the transition diagram, the language of a DFA is the set of labels along all the paths that lead from the start state to any accepting state.

Now, we need to make the notion of the language of a DFA precise. To do so, we define an ***extended transition function*** that describes what happens when we start in any state and follow any sequence of inputs. If δ is our transition function, then the extended transition function constructed from δ will be called $\hat{\delta}$. The extended transition function is a function that takes a state q and a string w and returns a state p — the state that the automaton reaches when starting in state q and processing the sequence of inputs w. We define $\hat{\delta}$ by induction on the length of the input string, as follows:

BASIS: $\hat{\delta}(q, \epsilon) = q$. That is, if we are in state q and read no inputs, then we are still in state q.

INDUCTION: Suppose w is a string of the form xa; that is, a is the last symbol of w, and x is the string consisting of all but the last symbol.[3] For example, $w = 1101$ is broken into $x = 110$ and $a = 1$. Then

$$\hat{\delta}(q, w) = \delta\big(\hat{\delta}(q, x), a\big) \tag{2.1}$$

Now (2.1) may seem like a lot to take in, but the idea is simple. To compute $\hat{\delta}(q, w)$, first compute $\hat{\delta}(q, x)$, the state that the automaton is in after processing all but the last symbol of w. Suppose this state is p; that is, $\hat{\delta}(q, x) = p$. Then $\hat{\delta}(q, w)$ is what we get by making a transition from state p on input a, the last symbol of w. That is, $\hat{\delta}(q, w) = \delta(p, a)$.

Example 2.4: Let us design a DFA to accept the language

$$L = \{w \mid w \text{ has both an even number of 0's and an even number of 1's}\}$$

It should not be surprising that the job of the states of this DFA is to count both the number of 0's and the number of 1's, but count them modulo 2. That is, the state is used to remember whether the number of 0's seen so far is even or odd, and also to remember whether the number of 1's seen so far is even or odd. There are thus four states, which can be given the following interpretations:

q_0: Both the number of 0's seen so far and the number of 1's seen so far are even.

q_1: The number of 0's seen so far is even, but the number of 1's seen so far is odd.

q_2: The number of 1's seen so far is even, but the number of 0's seen so far is odd.

q_3: Both the number of 0's seen so far and the number of 1's seen so far are odd.

State q_0 is both the start state and the lone accepting state. It is the start state, because before reading any inputs, the numbers of 0's and 1's seen so far are both zero, and zero is even. It is the only accepting state, because it describes exactly the condition for a sequence of 0's and 1's to be in language L.

We now know almost how to specify the DFA for language L. It is

$$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

[3]Recall our convention that letters at the beginning of the alphabet are symbols, and those near the end of the alphabet are strings. We need that convention to make sense of the phrase "of the form xa."

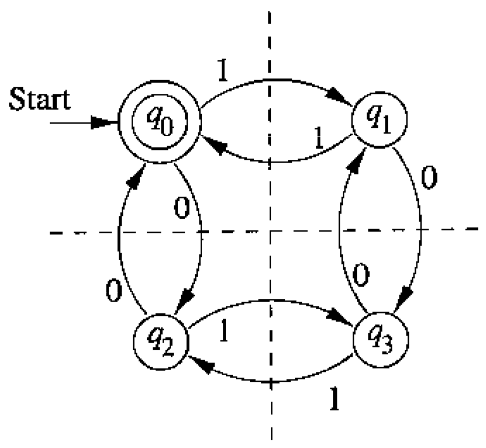


Figure 2.6: Transition diagram for the DFA of Example 2.4

where the transition function δ is described by the transition diagram of Fig. 2.6. Notice how each input 0 causes the state to cross the horizontal, dashed line. Thus, after seeing an even number of 0's we are always above the line, in state q_0 or q_1 while after seeing an odd number of 0's we are always below the line, in state q_2 or q_3. Likewise, every 1 causes the state to cross the vertical, dashed line. Thus, after seeing an even number of 1's, we are always to the left, in state q_0 or q_2, while after seeing an odd number of 1's we are to the right, in state q_1 or q_3. These observations are an informal proof that the four states have the interpretations attributed to them. However, one could prove the correctness of our claims about the states formally, by a mutual induction in the spirit of Example 1.23.

We can also represent this DFA by a transition table. Figure 2.7 shows this table. However, we are not just concerned with the design of this DFA; we want to use it to illustrate the construction of $\hat{\delta}$ from its transition function δ. Suppose the input is 110101. Since this string has even numbers of 0's and 1's both, we expect it is in the language. Thus, we expect that $\hat{\delta}(q_0, 110101) = q_0$, since q_0 is the only accepting state. Let us now verify that claim.

	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Figure 2.7: Transition table for the DFA of Example 2.4

The check involves computing $\hat{\delta}(q_0, w)$ for each prefix w of 110101, starting at ϵ and going in increasing size. The summary of this calculation is:

Standard Notation and Local Variables

After reading this section, you might imagine that our customary notation is required; that is, you *must* use δ for the transition function, use A for the name of a DFA, and so on. We tend to use the same variables to denote the same thing across all examples, because it helps to remind you of the types of variables, much the way a variable i in a program is almost always of integer type. However, we are free to call the components of an automaton, or anything else, anything we wish. Thus, you are free to call a DFA M and its transition function T if you like.

Moreover, you should not be surprised that the same variable means different things in different contexts. For example, the DFA's of Examples 2.1 and 2.4 both were given a transition function called δ. However, the two transition functions are each local variables, belonging only to their examples. These two transition functions are very different and bear no relationship to one another.

- $\hat{\delta}(q_0, \epsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta\big(\hat{\delta}(q_0, \epsilon), 1\big) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta\big(\hat{\delta}(q_0, 1), 1\big) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta\big(\hat{\delta}(q_0, 11), 0\big) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0, 1101) = \delta\big(\hat{\delta}(q_0, 110), 1\big) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta\big(\hat{\delta}(q_0, 1101), 0\big) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta\big(\hat{\delta}(q_0, 11010), 1\big) = \delta(q_1, 1) = q_0$.

□

2.2.5 The Language of a DFA

Now, we can define the *language* of a DFA $A = (Q, \Sigma, \delta, q_0, F)$. This language is denoted $L(A)$, and is defined by

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ is in } F\}$$

That is, the language of A is the set of strings w that take the start state q_0 to one of the accepting states. If L is $L(A)$ for some DFA A, then we say L is a *regular language*.

Example 2.5 : As we mentioned earlier, if A is the DFA of Example 2.1, then $L(A)$ is the set of all strings of 0's and 1's that contain a substring 01. If A is instead the DFA of Example 2.4, then $L(A)$ is the set of all strings of 0's and 1's whose numbers of 0's and 1's are both even. □

2.2.6 Exercises for Section 2.2

Exercise 2.2.1 : In Fig. 2.8 is a marble-rolling toy. A marble is dropped at A or B. Levers x_1, x_2, and x_3 cause the marble to fall either to the left or to the right. Whenever a marble encounters a lever, it causes the lever to reverse after the marble passes, so the next marble will take the opposite branch.

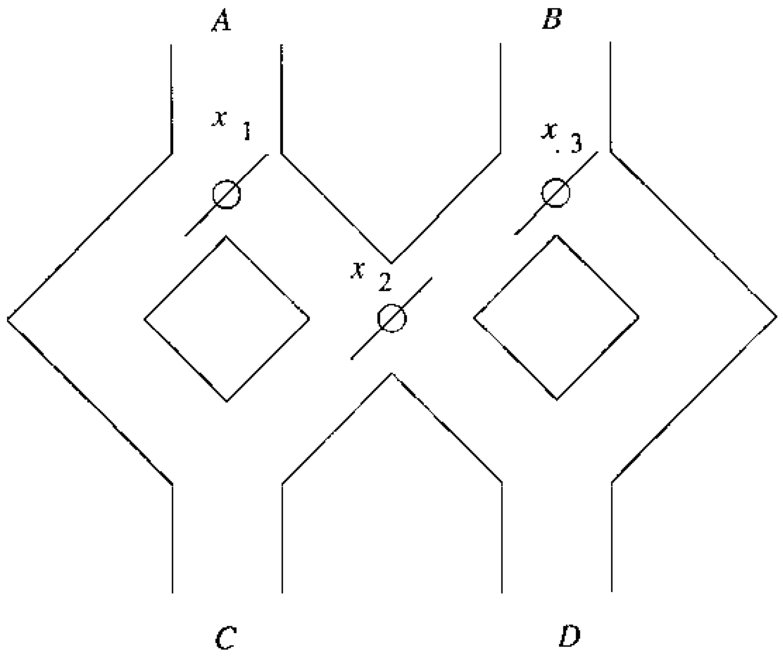


Figure 2.8: A marble-rolling toy

* a) Model this toy by a finite automaton. Let the inputs A and B represent the input into which the marble is dropped. Let acceptance correspond to the marble exiting at D; nonacceptance represents a marble exiting at C.

! b) Informally describe the language of the automaton.

c) Suppose that instead the levers switched *before* allowing the marble to pass. How would your answers to parts (a) and (b) change?

*! **Exercise 2.2.2 :** We defined $\hat{\delta}$ by breaking the input string into any string followed by a single symbol (in the inductive part, Equation 2.1). However, we informally think of $\hat{\delta}$ as describing what happens along a path with a certain

string of labels, and if so, then it should not matter how we break the input string in the definition of $\hat{\delta}$. Show that in fact, $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ for any state q and strings x and y. *Hint*: Perform an induction on $|y|$.

! **Exercise 2.2.3:** Show that for any state q, string x, and input symbol a, $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$. *Hint*: Use Exercise 2.2.2.

Exercise 2.2.4: Give DFA's accepting the following languages over the alphabet $\{0, 1\}$:

* a) The set of all strings ending in 00.

b) The set of all strings with three consecutive 0's (not necessarily at the end).

c) The set of strings with 011 as a substring.

! **Exercise 2.2.5:** Give DFA's accepting the following languages over the alphabet $\{0, 1\}$:

a) The set of all strings such that each block of five consecutive symbols contains at least two 0's.

b) The set of all strings whose tenth symbol from the right end is a 1.

c) The set of strings that either begin or end (or both) with 01.

d) The set of strings such that the number of 0's is divisible by five, and the number of 1's is divisible by 3.

!! **Exercise 2.2.6:** Give DFA's accepting the following languages over the alphabet $\{0, 1\}$:

* a) The set of all strings beginning with a 1 that, when interpreted as a binary integer, is a multiple of 5. For example, strings 101, 1010, and 1111 are in the language; 0, 100, and 111 are not.

b) The set of all strings that, when interpreted *in reverse* as a binary integer, is divisible by 5. Examples of strings in the language are 0, 10011, 1001100, and 0101.

Exercise 2.2.7: Let A be a DFA and q a particular state of A, such that $\delta(q, a) = q$ for all input symbols a. Show by induction on the length of the input that for all input strings w, $\hat{\delta}(q, w) = q$.

Exercise 2.2.8: Let A be a DFA and a a particular input symbol of A, such that for all states q of A we have $\delta(q, a) = q$.

a) Show by induction on n that for all $n \geq 0$, $\hat{\delta}(q, a^n) = q$, where a^n is the string consisting of n a's.

b) Show that either $\{a\}^* \subseteq L(A)$ or $\{a\}^* \cap L(A) = \emptyset$.

*! **Exercise 2.2.9:** Let $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ be a DFA, and suppose that for all a in Σ we have $\delta(q_0, a) = \delta(q_f, a)$.

a) Show that for all $w \neq \epsilon$ we have $\hat{\delta}(q_0, w) = \hat{\delta}(q_f, w)$.

b) Show that if x is a nonempty string in $L(A)$, then for all $k > 0$, x^k (i.e., x written k times) is also in $L(A)$.

*! **Exercise 2.2.10:** Consider the DFA with the following transition table:

	0	1
$\rightarrow A$	A	B
$*B$	B	A

Informally describe the language accepted by this DFA, and prove by induction on the length of an input string that your description is correct. *Hint*: When setting up the inductive hypothesis, it is wise to make a statement about what inputs get you to each state, not just what inputs get you to the accepting state.

! **Exercise 2.2.11:** Repeat Exercise 2.2.10 for the following transition table:

	0	1
$\rightarrow *A$	B	A
$*B$	C	A
C	C	C

2.3 Nondeterministic Finite Automata

A "nondeterministic" finite automaton (*NFA*) has the power to be in several states at once. This ability is often expressed as an ability to "guess" something about its input. For instance, when the automaton is used to search for certain sequences of characters (e.g., keywords) in a long text string, it is helpful to "guess" that we are at the beginning of one of those strings and use a sequence of states to do nothing but check that the string appears, character by character. We shall see an example of this type of application in Section 2.4.

Before examining applications, we need to define nondeterministic finite automata and show that each one accepts a language that is also accepted by some DFA. That is, the NFA's accept exactly the regular languages, just as DFA's do. However, there are reasons to think about NFA's. They are often more succinct and easier to design than DFA's. Moreover, while we can always convert an NFA to a DFA, the latter may have exponentially more states than the NFA; fortunately, cases of this type are rare.

2.3.1 An Informal View of Nondeterministic Finite Automata

Like the DFA, an NFA has a finite set of states, a finite set of input symbols, one start state and a set of accepting states. It also has a transition function, which we shall commonly call δ. The difference between the DFA and the NFA is in the type of δ. For the NFA, δ is a function that takes a state and input symbol as arguments (like the DFA's transition function), but returns a set of zero, one, or more states (rather than returning exactly one state, as the DFA must). We shall start with an example of an NFA, and then make the definitions precise.

Example 2.6: Figure 2.9 shows a nondeterministic finite automaton, whose job is to accept all and only the strings of 0's and 1's that end in 01. State q_0 is the start state, and we can think of the automaton as being in state q_0 (perhaps among other states) whenever it has not yet "guessed" that the final 01 has begun. It is always possible that the next symbol does not begin the final 01, even if that symbol is 0. Thus, state q_0 may transition to itself on both 0 and 1.

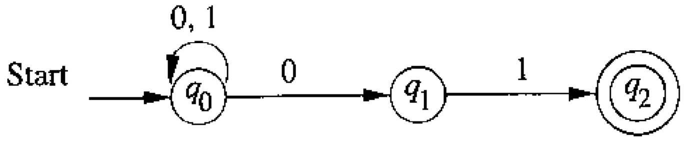


Figure 2.9: An NFA accepting all strings that end in 01

However, if the next symbol is 0, this NFA also guesses that the final 01 has begun. An arc labeled 0 thus leads from q_0 to state q_1. Notice that there are two arcs labeled 0 out of q_0. The NFA has the option of going either to q_0 or to q_1, and in fact it does both, as we shall see when we make the definitions precise. In state q_1, the NFA checks that the next symbol is 1, and if so, it goes to state q_2 and accepts.

Notice that there is no arc out of q_1 labeled 0, and there are no arcs at all out of q_2. In these situations, the thread of the NFA's existence corresponding to those states simply "dies," although other threads may continue to exist. While a DFA has exactly one arc out of each state for each input symbol, an NFA has no such constraint; we have seen in Fig. 2.9 cases where the number of arcs is zero, one, and two, for example.

Figure 2.10 suggests how an NFA processes inputs. We have shown what happens when the automaton of Fig. 2.9 receives the input sequence 00101. It starts in only its start state, q_0. When the first 0 is read, the NFA may go to either state q_0 or state q_1, so it does both. These two threads are suggested by the second column in Fig. 2.10.

Then, the second 0 is read. State q_0 may again go to both q_0 and q_1. However, state q_1 has no transition on 0, so it "dies." When the third input, a

$$\begin{tikzcd}
q_0 \arrow[r] \arrow[dr] & q_0 \arrow[r] \arrow[dr] & q_0 \arrow[r] & q_0 \arrow[r] \arrow[dr] & q_0 \arrow[r] & q_0 \\
 & q_1 & q_1 \arrow[dr] & & q_1 \arrow[dr] & \\
 & \text{(stuck)} & & q_2 & & q_2 \\
 & & & \text{(stuck)} & & \\
0 & 0 & 1 & 0 & 1 &
\end{tikzcd}$$

Figure 2.10: The states an NFA is in during the processing of input sequence 00101

1, occurs, we must consider transitions from both q_0 and q_1. We find that q_0 goes only to q_0 on 1, while q_1 goes only to q_2. Thus, after reading 001, the NFA is in states q_0 and q_2. Since q_2 is an accepting state, the NFA accepts 001.

However, the input is not finished. The fourth input, a 0, causes q_2's thread to die, while q_0 goes to both q_0 and q_1. The last input, a 1, sends q_0 to q_0 and q_1 to q_2. Since we are again in an accepting state, 00101 is accepted. □

2.3.2 Definition of Nondeterministic Finite Automata

Now, let us introduce the formal notions associated with nondeterministic finite automata. The differences between DFA's and NFA's will be pointed out as we do. An NFA is represented essentially like a DFA:

$$A = (Q, \Sigma, \delta, q_0, F)$$

where:

1. Q is a finite set of *states*.

2. Σ is a finite set of *input symbols*.

3. q_0, a member of Q, is the *start state*.

4. F, a subset of Q, is the set of *final* (or *accepting*) states.

5. δ, the *transition function* is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q. Notice that the only difference between an NFA and a DFA is in the type of value that δ returns: a set of states in the case of an NFA and a single state in the case of a DFA.

Example 2.7: The NFA of Fig. 2.9 can be specified formally as

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

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Figure 2.11: Transition table for an NFA that accepts all strings ending in 01

where the transition function δ is given by the transition table of Fig. 2.11. □

Notice that transition tables can be used to specify the transition function for an NFA as well as for a DFA. The only difference is that each entry in the table for the NFA is a set, even if the set is a *singleton* (has one member). Also notice that when there is no transition at all from a given state on a given input symbol, the proper entry is $\emptyset$, the empty set.

2.3.3 The Extended Transition Function

As for DFA's, we need to extend the transition function δ of an NFA to a function $\hat{\delta}$ that takes a state q and a string of input symbols w, and returns the set of states that the NFA is in if it starts in state q and processes the string w. The idea was suggested by Fig. 2.10; in essence $\hat{\delta}(q, w)$ is the column of states found after reading w, if q is the lone state in the first column. For instance, Fig. 2.10 suggests that $\hat{\delta}(q_0, 001) = \{q_0, q_2\}$. Formally, we define $\hat{\delta}$ for an NFA's transition function δ by:

BASIS: $\hat{\delta}(q, \epsilon) = \{q\}$. That is, without reading any input symbols, we are only in the state we began in.

INDUCTION: Suppose w is of the form $w = xa$, where a is the final symbol of w and x is the rest of w. Also suppose that $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$. Let

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$$

Then $\hat{\delta}(q, w) = \{r_1, r_2, \ldots, r_m\}$. Less formally, we compute $\hat{\delta}(q, w)$ by first computing $\hat{\delta}(q, x)$, and by then following any transition from any of these states that is labeled a.

Example 2.8: Let us use $\hat{\delta}$ to describe the processing of input 00101 by the NFA of Fig. 2.9. A summary of the steps is:

1. $\hat{\delta}(q_0, \epsilon) = \{q_0\}$.

2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$.

3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$.

4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$.

6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

Line (1) is the basis rule. We obtain line (2) by applying δ to the lone state, q_0, that is in the previous set, and get $\{q_0, q_1\}$ as a result. Line (3) is obtained by taking the union over the two states in the previous set of what we get when we apply δ to them with input 0. That is, $\delta(q_0, 0) = \{q_0, q_1\}$, while $\delta(q_1, 0) = \emptyset$. For line (4), we take the union of $\delta(q_0, 1) = \{q_0\}$ and $\delta(q_1, 1) = \{q_2\}$. Lines (5) and (6) are similar to lines (3) and (4). □

2.3.4 The Language of an NFA

As we have suggested, an NFA accepts a string w if it is possible to make any sequence of choices of next state, while reading the characters of w, and go from the start state to any accepting state. The fact that other choices using the input symbols of w lead to a nonaccepting state, or do not lead to any state at all (i.e., the sequence of states "dies"), does not prevent w from being accepted by the NFA as a whole. Formally, if $A = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

That is, $L(A)$ is the set of strings w in Σ^* such that $\hat{\delta}(q_0, w)$ contains at least one accepting state.

Example 2.9: As an example, let us prove formally that the NFA of Fig. 2.9 accepts the language $L = \{w \mid w \text{ ends in } 01\}$. The proof is a mutual induction of the following three statements that characterize the three states:

1. $\hat{\delta}(q_0, w)$ contains q_0 for every w.

2. $\hat{\delta}(q_0, w)$ contains q_1 if and only if w ends in 0.

3. $\hat{\delta}(q_0, w)$ contains q_2 if and only if w ends in 01.

To prove these statements, we need to consider how A can reach each state; i.e., what was the last input symbol, and in what state was A just before reading that symbol?

Since the language of this automaton is the set of strings w such that $\hat{\delta}(q_0, w)$ contains q_2 (because q_2 is the only accepting state), the proof of these three statements, in particular the proof of (3), guarantees that the language of this NFA is the set of strings ending in 01. The proof of the theorem is an induction on $|w|$, the length of w, starting with length 0.

BASIS: If $|w| = 0$, then $w = \epsilon$. Statement (1) says that $\hat{\delta}(q_0, \epsilon)$ contains q_0, which it does by the basis part of the definition of $\hat{\delta}$. For statement (2), we know that ϵ does not end in 0, and we also know that $\hat{\delta}(q_0, \epsilon)$ does not contain q_1, again by the basis part of the definition of $\hat{\delta}$. Thus, the hypotheses of both directions of the if-and-only-if statement are false, and therefore both directions of the statement are true. The proof of statement (3) for $w = \epsilon$ is essentially the same as the above proof for statement (2).

INDUCTION: Assume that $w = xa$, where a is a symbol, either 0 or 1. We may assume statements (1) through (3) hold for x, and we need to prove them for w. That is, we assume $|w| = n + 1$, so $|x| = n$. We assume the inductive hypothesis for n and prove it for $n + 1$.

1. We know that $\hat{\delta}(q_0, x)$ contains q_0. Since there are transitions on both 0 and 1 from q_0 to itself, it follows that $\hat{\delta}(q_0, w)$ also contains q_0, so statement (1) is proved for w.

2. (If) Assume that w ends in 0; i.e., $a = 0$. By statement (1) applied to x, we know that $\hat{\delta}(q_0, x)$ contains q_0. Since there is a transition from q_0 to q_1 on input 0, we conclude that $\hat{\delta}(q_0, w)$ contains q_1.

   (Only-if) Suppose $\hat{\delta}(q_0, w)$ contains q_1. If we look at the diagram of Fig. 2.9, we see that the only way to get into state q_1 is if the input sequence w is of the form $x0$. That is enough to prove the "only-if" portion of statement (2).

3. (If) Assume that w ends in 01. Then if $w = xa$, we know that $a = 1$ and x ends in 0. By statement (2) applied to x, we know that $\hat{\delta}(q_0, x)$ contains q_1. Since there is a transition from q_1 to q_2 on input 1, we conclude that $\hat{\delta}(q_0, w)$ contains q_2.

   (Only-if) Suppose $\hat{\delta}(q_0, w)$ contains q_2. Looking at the diagram of Fig. 2.9, we discover that the only way to get to state q_2 is for w to be of the form $x1$, where $\hat{\delta}(q_0, x)$ contains q_1. By statement (2) applied to x, we know that x ends in 0. Thus, w ends in 01, and we have proved statement (3).

□

2.3.5 Equivalence of Deterministic and Nondeterministic Finite Automata

Although there are many languages for which an NFA is easier to construct than a DFA, such as the language (Example 2.6) of strings that end in 01, it is a surprising fact that every language that can be described by some NFA can also be described by some DFA. Moreover, the DFA in practice has about as many states as the NFA, although it often has more transitions. In the worst case, however, the smallest DFA can have 2^n states while the smallest NFA for the same language has only n states.

The proof that DFA's can do whatever NFA's can do involves an important "construction" called the ***subset construction*** because it involves constructing all subsets of the set of states of the NFA. In general, many proofs about automata involve constructing one automaton from another. It is important for us to observe the subset construction as an example of how one formally describes one automaton in terms of the states and transitions of another, without knowing the specifics of the latter automaton.

The subset construction starts from an NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Its goal is the description of a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(D) = L(N)$. Notice that the input alphabets of the two automata are the same, and the start state of D is the set containing only the start state of N. The other components of D are constructed as follows.

- Q_D is the set of subsets of Q_N; i.e., Q_D is the ***power set*** of Q_N. Note that if Q_N has n states, then Q_D will have 2^n states. Often, not all these states are accessible from the start state of Q_D. Inaccessible states can be "thrown away," so effectively, the number of states of D may be much smaller than 2^n.

- F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N's states that include at least one accepting state of N.

- For each set $S \subseteq Q_N$ and for each input symbol a in Σ,

$$\delta_D(S, a) = \bigcup_{p \text{ in } S} \delta_N(p, a)$$

  That is, to compute $\delta_D(S, a)$ we look at all the states p in S, see what states N goes to from p on input a, and take the union of all those states.

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Figure 2.12: The complete subset construction from Fig. 2.9

Example 2.10: Let N be the automaton of Fig. 2.9 that accepts all strings that end in 01. Since N's set of states is $\{q_0, q_1, q_2\}$, the subset construction produces a DFA with $2^3 = 8$ states, corresponding to all the subsets of these three states. Figure 2.12 shows the transition table for these eight states; we shall show shortly the details of how some of these entries are computed.

Notice that this transition table belongs to a deterministic finite automaton. Even though the entries in the table are sets, the states of the constructed DFA *are* sets. To make the point clearer, we can invent new names for these states, e.g., A for $\emptyset$, B for $\{q_0\}$, and so on. The DFA transition table of Fig 2.13 defines exactly the same automaton as Fig. 2.12, but makes clear the point that the entries in the table are single states of the DFA.

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Figure 2.13: Renaming the states of Fig. 2.12

Of the eight states in Fig. 2.13, starting in the start state B, we can only reach states B, E, and F. The other five states are inaccessible from the start state and may as well not be there. We often can avoid the exponential-time step of constructing transition-table entries for every subset of states if we perform "lazy evaluation" on the subsets, as follows.

BASIS: We know for certain that the singleton set consisting only of N's start state is accessible.

INDUCTION: Suppose we have determined that set S of states is accessible. Then for each input symbol a, compute the set of states $\delta_D(S, a)$; we know that these sets of states will also be accessible.

For the example at hand, we know that $\{q_0\}$ is a state of the DFA D. We find that $\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ and $\delta_D(\{q_0\}, 1) = \{q_0\}$. Both these facts are established by looking at the transition diagram of Fig. 2.9 and observing that on 0 there are arcs out of q_0 to both q_0 and q_1, while on 1 there is an arc only to q_0. We thus have one row of the transition table for the DFA: the second row in Fig. 2.12.

One of the two sets we computed is "old"; $\{q_0\}$ has already been considered. However, the other — $\{q_0, q_1\}$ — is new and its transitions must be computed. We find $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ and $\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$. For instance, to see the latter calculation, we know that

$$\delta_D(\{q_0, q_1\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$$

We now have the fifth row of Fig. 2.12, and we have discovered one new state of D, which is $\{q_0, q_2\}$. A similar calculation tells us

$$\delta_D(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$$
$$\delta_D(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \emptyset = \{q_0\}$$

These calculations give us the sixth row of Fig. 2.12, but it gives us only sets of states that we have already seen.

Thus, the subset construction has converged; we know all the accessible states and their transitions. The entire DFA is shown in Fig. 2.14. Notice that it has only three states, which is, by coincidence, exactly the same number of states as the NFA of Fig. 2.9, from which it was constructed. However, the DFA of Fig. 2.14 has six transitions, compared with the four transitions in Fig. 2.9. □

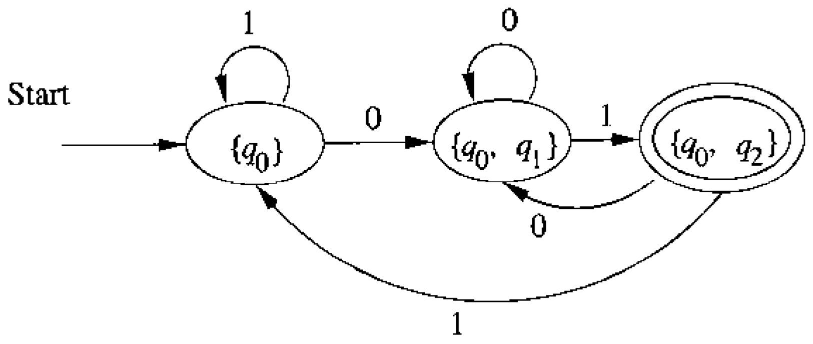


Figure 2.14: The DFA constructed from the NFA of Fig 2.9

We need to show formally that the subset construction works, although the intuition was suggested by the examples. After reading sequence of input symbols w, the constructed DFA is in one state that is the set of NFA states that the NFA would be in after reading w. Since the accepting states of the DFA are those sets that include at least one accepting state of the NFA, and the NFA also accepts if it gets into at least one of its accepting states, we may then conclude that the DFA and NFA accept exactly the same strings, and therefore accept the same language.

Theorem 2.11: If $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by the subset construction, then $L(D) = L(N)$.

PROOF: What we actually prove first, by induction on $|w|$, is that

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Notice that each of the $\hat{\delta}$ functions returns a set of states from Q_N, but $\hat{\delta}_D$ interprets this set as one of the states of Q_D (which is the power set of Q_N), while $\hat{\delta}_N$ interprets this set as a subset of Q_N.

BASIS: Let $|w| = 0$; that is, $w = \epsilon$. By the basis definitions of $\hat{\delta}$ for DFA's and NFA's, both $\hat{\delta}_D(\{q_0\}, \epsilon)$ and $\hat{\delta}_N(q_0, \epsilon)$ are $\{q_0\}$.

INDUCTION: Let w be of length $n + 1$, and assume the statement for length n. Break w up as $w = xa$, where a is the final symbol of w. By the inductive hypothesis, $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Let both these sets of N's states be $\{p_1, p_2, \ldots, p_k\}$.

The inductive part of the definition of $\hat{\delta}$ for NFA's tells us

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

The subset construction, on the other hand, tells us that

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

Now, let us use (2.3) and the fact that $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ in the inductive part of the definition of $\hat{\delta}$ for DFA's:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D\big(\hat{\delta}_D(\{q_0\}, x), a\big) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.4}$$

Thus, Equations (2.2) and (2.4) demonstrate that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. When we observe that D and N both accept w if and only if $\hat{\delta}_D(\{q_0\}, w)$ or $\hat{\delta}_N(q_0, w)$, respectively, contain a state in F_N, we have a complete proof that $L(D) = L(N)$. $\square$

Theorem 2.12: A language L is accepted by some DFA if and only if L is accepted by some NFA.

PROOF: (If) The "if" part is the subset construction and Theorem 2.11.

(Only-if) This part is easy; we have only to convert a DFA into an identical NFA. Put intuitively, if we have the transition diagram for a DFA, we can also interpret it as the transition diagram of an NFA, which happens to have exactly one choice of transition in any situation. More formally, let $D = (Q, \Sigma, \delta_D, q_0, F)$ be a DFA. Define $N = (Q, \Sigma, \delta_N, q_0, F)$ to be the equivalent NFA, where δ_N is defined by the rule:

- If $\delta_D(q, a) = p$, then $\delta_N(q, a) = \{p\}$.

It is then easy to show by induction on $|w|$, that if $\hat{\delta}_D(q_0, w) = p$ then

$$\hat{\delta}_N(q_0, w) = \{p\}$$

We leave the proof to the reader. As a consequence, w is accepted by D if and only if it is accepted by N; i.e., $L(D) = L(N)$. $\square$

2.3.6 A Bad Case for the Subset Construction

In Example 2.10 we found that the DFA had no more states than the NFA. As we mentioned, it is quite common in practice for the DFA to have roughly the same number of states as the NFA from which it is constructed. However, exponential growth in the number of states is possible; all the 2^n DFA states that we could construct from an n-state NFA may turn out to be accessible. The following example does not quite reach that bound, but it is an understandable way to reach 2^n states in the smallest DFA that is equivalent to an $n + 1$-state NFA.

Example 2.13 : Consider the NFA N of Fig. 2.15. $L(N)$ is the set of all strings of 0's and 1's such that the nth symbol from the end is 1. Intuitively, a DFA D that accepts this language must remember the last n symbols it has read. Since any of 2^n subsets of the last n symbols could have been 1, if D has fewer than 2^n states, then there would be some state q such that D can be in state q after reading two different sequences of n bits, say $a_1a_2\cdots a_n$ and $b_1b_2\cdots b_n$.

Since the sequences are different, they must differ in some position, say $a_i \neq b_i$. Suppose (by symmetry) that $a_i = 1$ and $b_i = 0$. If $i = 1$, then q must be both an accepting state and a nonaccepting state, since $a_1a_2\cdots a_n$ is accepted (the nth symbol from the end is 1) and $b_1b_2\cdots b_n$ is not. If $i > 1$, then consider the state p that D enters after reading $i - 1$ 0's. Then p must be both accepting and nonaccepting, since $a_ia_{i+1}\cdots a_n00\cdots0$ is accepted and $b_ib_{i+1}\cdots b_n00\cdots0$ is not.

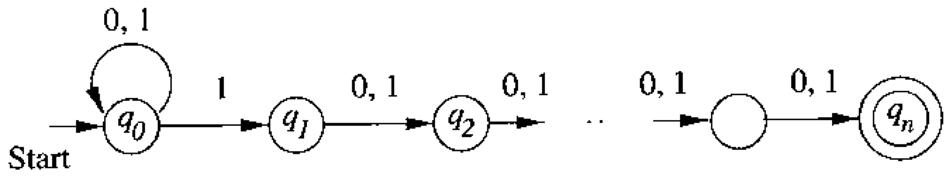


Figure 2.15: This NFA has no equivalent DFA with fewer than 2^n states

Now, let us see how the NFA N of Fig. 2.15 works. There is a state q_0 that the NFA is always in, regardless of what inputs have been read. If the next input is 1, N may also "guess" that this 1 will be the nth symbol from the end, so it goes to state q_1 as well as q_0. From state q_1, any input takes N to q_2, the next input takes it to q_3, and so on, until $n - 1$ inputs later, it is in the accepting state q_n. The formal statement of what the states of N do is:

1. N is in state q_0 after reading any sequence of inputs w.
2. N is in state q_i, for $i = 1, 2, \ldots, n$, after reading input sequence w if and only if the ith symbol from the end of w is 1; that is, w is of the form $x1a_1a_2\cdots a_{i-1}$, where the a_j's are each input symbols.

We shall not prove these statements formally; the proof is an easy induction on $|w|$, mimicking Example 2.9. To complete the proof that the automaton

The Pigeonhole Principle

In Example 2.13 we used an important reasoning technique called the *pigeonhole principle*. Colloquially, if you have more pigeons than pigeonholes, and each pigeon flies into some pigeonhole, then there must be at least one hole that has more than one pigeon. In our example, the "pigeons" are the sequences of n bits, and the "pigeonholes" are the states. Since there are fewer states than sequences, one state must be assigned two sequences.

The pigeonhole principle may appear obvious, but it actually depends on the number of pigeonholes being finite. Thus, it works for finite-state automata, with the states as pigeonholes, but does not apply to other kinds of automata that have an infinite number of states.

To see why the finiteness of the number of pigeonholes is essential, consider the infinite situation where the pigeonholes correspond to integers $1, 2, \ldots$. Number the pigeons $0, 1, 2, \ldots$, so there is one more pigeon than there are pigeonholes. However, we can send pigeon i to hole $i + 1$ for all $i \geq 0$. Then each of the infinite number of pigeons gets a pigeonhole, and no two pigeons have to share a pigeonhole.

accepts exactly those strings with a 1 in the nth position from the end, we consider statement (2) with $i = n$. That says N is in state q_n if and only if the nth symbol from the end is 1. But q_n is the only accepting state, so that condition also characterizes exactly the set of strings accepted by N. □

2.3.7 Exercises for Section 2.3

* **Exercise 2.3.1:** Convert to a DFA the following NFA:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
$*s$	$\{s\}$	$\{s\}$

Exercise 2.3.2: Convert to a DFA the following NFA:

	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\emptyset$	$\{p\}$

Dead States and DFA's Missing Some Transitions

We have formally defined a DFA to have a transition from any state, on any input symbol, to exactly one state. However, sometimes, it is more convenient to design the DFA to "die" in situations where we know it is impossible for any extension of the input sequence to be accepted. For instance, observe the automaton of Fig. 1.2, which did its job by recognizing a single keyword, **then**, and nothing else. Technically, this automaton is not a DFA, because it lacks transitions on most symbols from each of its states.

However, such an automaton is an NFA. If we use the subset construction to convert it to a DFA, the automaton looks almost the same, but it includes a *dead state*, that is, a nonaccepting state that goes to itself on every possible input symbol. The dead state corresponds to $\emptyset$, the empty set of states of the automaton of Fig. 1.2.

In general, we can add a dead state to any automaton that has *no more* than one transition for any state and input symbol. Then, add a transition to the dead state from each other state q, on all input symbols for which q has no other transition. The result will be a DFA in the strict sense. Thus, we shall sometimes refer to an automaton as a DFA if it has *at most* one transition out of any state on any symbol, rather than if it has *exactly one* transition.

! **Exercise 2.3.3:** Convert the following NFA to a DFA and informally describe the language it accepts.

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r, s\}$	$\{t\}$
r	$\{p, r\}$	$\{t\}$
$*s$	$\emptyset$	$\emptyset$
$*t$	$\emptyset$	$\emptyset$

! **Exercise 2.3.4:** Give nondeterministic finite automata to accept the following languages. Try to take advantage of nondeterminism as much as possible.

* a) The set of strings over alphabet $\{0, 1, \ldots, 9\}$ such that the final digit has appeared before.

b) The set of strings over alphabet $\{0, 1, \ldots, 9\}$ such that the final digit has *not* appeared before.

c) The set of strings of 0's and 1's such that there are two 0's separated by a number of positions that is a multiple of 4. Note that 0 is an allowable multiple of 4.

Exercise 2.3.5: In the only-if portion of Theorem 2.12 we omitted the proof by induction on $|w|$ that if $\hat{\delta}_D(q_0, w) = p$ then $\hat{\delta}_N(q_0, w) = \{p\}$. Supply this proof.

! **Exercise 2.3.6:** In the box on "Dead States and DFA's Missing Some Transitions," we claim that if N is an NFA that has at most one choice of state for any state and input symbol (i.e., $\delta(q, a)$ never has size greater than 1), then the DFA D constructed from N by the subset construction has exactly the states and transitions of N plus transitions to a new dead state whenever N is missing a transition for a given state and input symbol. Prove this contention.

Exercise 2.3.7: In Example 2.13 we claimed that the NFA N is in state q_i, for $i = 1, 2, \ldots, n$, after reading input sequence w if and only if the ith symbol from the end of w is 1. Prove this claim.

2.4 An Application: Text Search

In this section, we shall see that the abstract study of the previous section, where we considered the "problem" of deciding whether a sequence of bits ends in 01, is actually an excellent model for several real problems that appear in applications such as Web search and extraction of information from text.

2.4.1 Finding Strings in Text

A common problem in the age of the Web and other on-line text repositories is the following. Given a set of words, find all documents that contain one (or all) of those words. A search engine is a popular example of this process. The search engine uses a particular technology, called *inverted indexes*, where for each word appearing on the Web (there are 100,000,000 different words), a list of all the places where that word occurs is stored. Machines with very large amounts of main memory keep the most common of these lists available, allowing many people to search for documents at once.

Inverted-index techniques do not make use of finite automata, but they also take very large amounts of time for crawlers to copy the Web and set up the indexes. There are a number of related applications that are unsuited for inverted indexes, but are good applications for automaton-based techniques. The characteristics that make an application suitable for searches that use automata are:

1. The repository on which the search is conducted is rapidly changing. For example:

   (a) Every day, news analysts want to search the day's on-line news articles for relevant topics. For example, a financial analyst might search for certain stock ticker symbols or names of companies.

(b) A "shopping robot" wants to search for the current prices charged for the items that its clients request. The robot will retrieve current catalog pages from the Web and then search those pages for words that suggest a price for a particular item.

2. The documents to be searched cannot be cataloged. For example, Amazon.com does not make it easy for crawlers to find all the pages for all the books that the company sells. Rather, these pages are generated "on the fly" in response to queries. However, we could send a query for books on a certain topic, say "finite automata," and then search the pages retrieved for certain words, e.g., "excellent" in a review portion.

2.4.2 Nondeterministic Finite Automata for Text Search

Suppose we are given a set of words, which we shall call the *keywords*, and we want to find occurrences of any of these words. In applications such as these, a useful way to proceed is to design a nondeterministic finite automaton, which signals, by entering an accepting state, that it has seen one of the keywords. The text of a document is fed, one character at a time to this NFA, which then recognizes occurrences of the keywords in this text. There is a simple form to an NFA that recognizes a set of keywords.

1. There is a start state with a transition to itself on every input symbol, e.g. every printable ASCII character if we are examining text. Intuitively, the start state represents a "guess" that we have not yet begun to see one of the keywords, even if we have seen some letters of one of these words.

2. For each keyword $a_1a_2\cdots a_k$, there are k states, say $q_1, q_2, \ldots, q_k$. There is a transition from the start state to q_1 on symbol a_1, a transition from q_1 to q_2 on symbol a_2, and so on. The state q_k is an accepting state and indicates that the keyword $a_1a_2\cdots a_k$ has been found.

Example 2.14: Suppose we want to design an NFA to recognize occurrences of the words web and ebay. The transition diagram for the NFA designed using the rules above is in Fig. 2.16. State 1 is the start state, and we use Σ to stand for the set of all printable ASCII characters. States 2 through 4 have the job of recognizing web, while states 5 through 8 recognize ebay. □

Of course the NFA is not a program. We have two major choices for an implementation of this NFA.

1. Write a program that simulates this NFA by computing the set of states it is in after reading each input symbol. The simulation was suggested in Fig. 2.10.

2. Convert the NFA to an equivalent DFA using the subset construction. Then simulate the DFA directly.

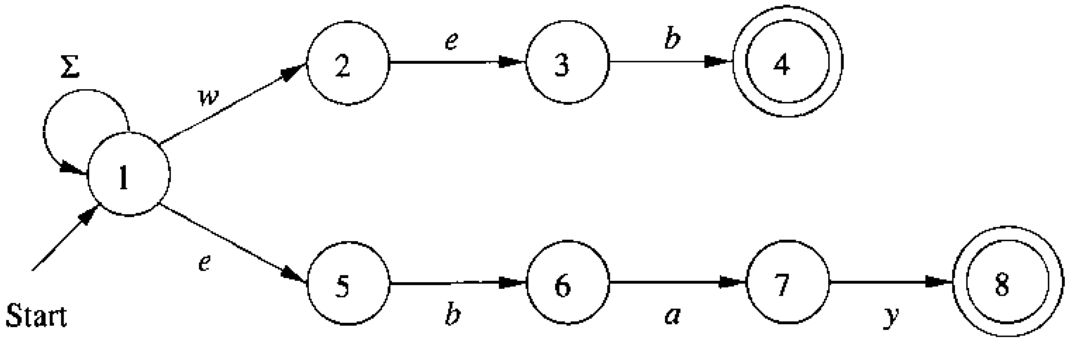


Figure 2.16: An NFA that searches for the words web and ebay

Some text-processing programs, such as advanced forms of the UNIX **grep** command (**egrep** and **fgrep**) actually use a mixture of these two approaches. However, for our purposes, conversion to a DFA is easy and is guaranteed not to increase the number of states.

2.4.3 A DFA to Recognize a Set of Keywords

We can apply the subset construction to any NFA. However, when we apply that construction to an NFA that was designed from a set of keywords, according to the strategy of Section 2.4.2, we find that the number of states of the DFA is never greater than the number of states of the NFA. Since in the worst case the number of states exponentiates as we go to the DFA, this observation is good news and explains why the method of designing an NFA for keywords and then constructing a DFA from it is used frequently. The rules for constructing the set of DFA states is as follows.

a) If q_0 is the start state of the NFA, then $\{q_0\}$ is one of the states of the DFA.

b) Suppose p is one of the NFA states, and it is reached from the start state along a path whose symbols are $a_1a_2\cdots a_m$. Then one of the DFA states is the set of NFA states consisting of:

   1. q_0.

   2. p.

   3. Every other state of the NFA that is reachable from q_0 by following a path whose labels are a suffix of $a_1a_2\cdots a_m$, that is, any sequence of symbols of the form $a_ja_{j+1}\cdots a_m$.

Note that in general, there will be one DFA state for each NFA state p. However, in step (b), two states may actually yield the same set of NFA states, and thus become one state of the DFA. For example, if two of the keywords begin with the same letter, say a, then the two NFA states that are reached from q_0 by an arc labeled a will yield the same set of NFA states and thus get merged in the DFA.

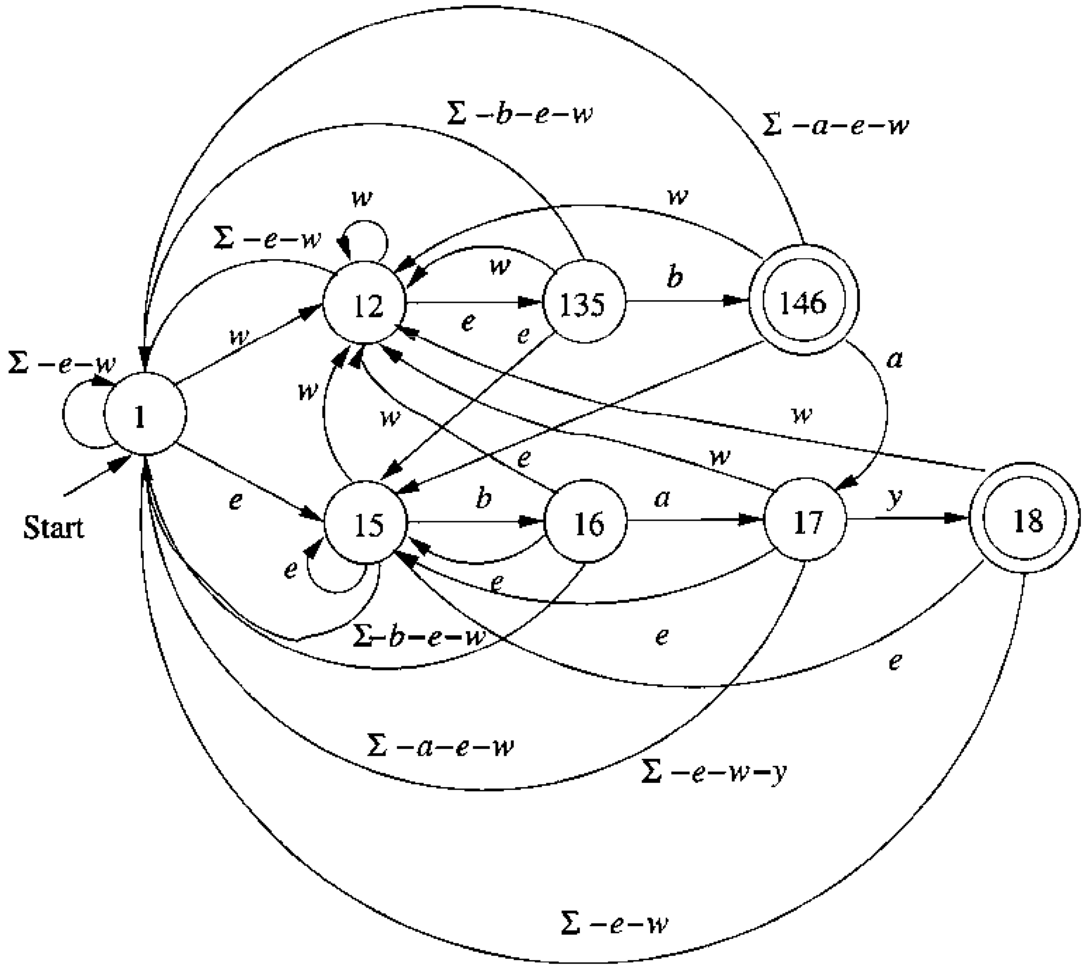


Figure 2.17: Conversion of the NFA from Fig. 2.16 to a DFA

Example 2.15: The construction of a DFA from the NFA of Fig. 2.16 is shown in Fig. 2.17. Each of the states of the DFA is located in the same position as the state p from which it is derived using rule (b) above. For example, consider the state 135, which is our shorthand for $\{1, 3, 5\}$. This state was constructed from state 3. It includes the start state, 1, because every set of the DFA states does. It also includes state 5 because that state is reached from state 1 by a suffix, e, of the string we that reaches state 3 in Fig. 2.16.

The transitions for each of the DFA states may be calculated according to the subset construction. However, the rule is simple. From any set of states that includes the start state q_0 and some other states $\{p_1, p_2, \ldots, p_n\}$, determine, for each symbol x, where the p_i's go in the NFA, and let this DFA state have a transition labeled x to the DFA state consisting of q_0 and all the targets of the

p_i's on symbol x. On all symbols x such that there are no transitions out of any of the p_i's on symbol x, let this DFA state have a transition on x to that state of the DFA consisting of q_0 and all states that are reached from q_0 in the NFA following an arc labeled x.

For instance, consider state 135 of Fig. 2.17. The NFA of Fig. 2.16 has transitions on symbol b from states 3 and 5 to states 4 and 6, respectively. Therefore, on symbol b, 135 goes to 146. On symbol e, there are no transitions of the NFA out of 3 or 5, but there is a transition from 1 to 5. Thus, in the DFA, 135 goes to 15 on input e. Similarly, on input w, 135 goes to 12.

On every other symbol x, there are no transitions out or 3 or 5, and state 1 goes only to itself. Thus, there are transitions from 135 to 1 on every symbol in Σ other than b, e, and w. We use the notation $\Sigma - b - e - w$ to represent this set, and use similar representations of other sets in which a few symbols are removed from Σ. □

2.4.4 Exercises for Section 2.4

Exercise 2.4.1: Design NFA's to recognize the following sets of strings.

* a) abc, abd, and aacd. Assume the alphabet is $\{a, b, c, d\}$.

b) 0101, 101, and 011.

c) ab, bc, and ca. Assume the alphabet is $\{a, b, c\}$.

Exercise 2.4.2: Convert each of your NFA's from Exercise 2.4.1 to DFA's.

2.5 Finite Automata With Epsilon-Transitions

We shall now introduce another extension of the finite automaton. The new "feature" is that we allow a transition on ϵ, the empty string. In effect, an NFA is allowed to make a transition spontaneously, without receiving an input symbol. Like the nondeterminism added in Section 2.3, this new capability does not expand the class of languages that can be accepted by finite automata, but it does give us some added "programming convenience." We shall also see, when we take up regular expressions in Section 3.1, how NFA's with ϵ-transitions, which we call *ϵ-NFA's*, are closely related to regular expressions and useful in proving the equivalence between the classes of languages accepted by finite automata and by regular expressions.

2.5.1 Uses of ϵ-Transitions

We shall begin with an informal treatment of ϵ-NFA's, using transition diagrams with ϵ allowed as a label. In the examples to follow, think of the automaton as accepting those sequences of labels along paths from the start state to an accepting state. However, each ϵ along a path is "invisible"; i.e., it contributes nothing to the string along the path.

Example 2.16: In Fig. 2.18 is an ϵ-NFA that accepts decimal numbers consisting of:

1. An optional $+$ or $-$ sign,

2. A string of digits,

3. A decimal point, and

4. Another string of digits. Either this string of digits, or the string (2) can be empty, but at least one of the two strings of digits must be nonempty.

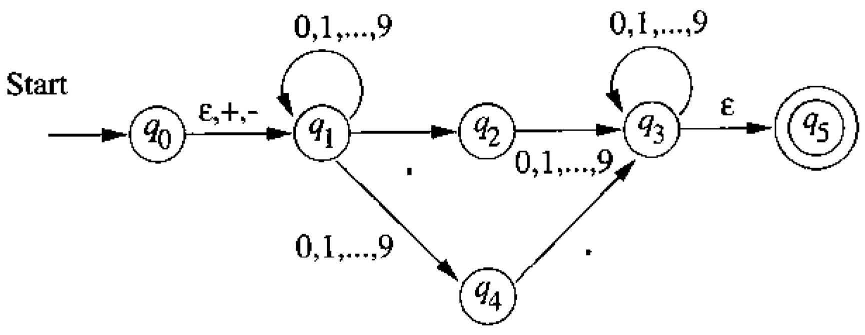


Figure 2.18: An ϵ-NFA accepting decimal numbers

Of particular interest is the transition from q_0 to q_1 on any of ϵ, $+$, or $-$. Thus, state q_1 represents the situation in which we have seen the sign if there is one, and perhaps some digits, but not the decimal point. State q_2 represents the situation where we have just seen the decimal point, and may or may not have seen prior digits. In q_4 we have definitely seen at least one digit, but not the decimal point. Thus, the interpretation of q_3 is that we have seen a decimal point and at least one digit, either before or after the decimal point. We may stay in q_3 reading whatever digits there are, and also have the option of "guessing" the string of digits is complete and going spontaneously to q_5, the accepting state. □

Example 2.17: The strategy we outlined in Example 2.14 for building an NFA that recognizes a set of keywords can be simplified further if we allow ϵ-transitions. For instance, the NFA recognizing the keywords web and ebay, which we saw in Fig. 2.16, can also be implemented with ϵ-transitions as in Fig. 2.19. In general, we construct a complete sequence of states for each keyword, as if it were the only word the automaton needed to recognize. Then, we add a new start state (state 9 in Fig. 2.19), with ϵ-transitions to the start-states of the automata for each of the keywords. □

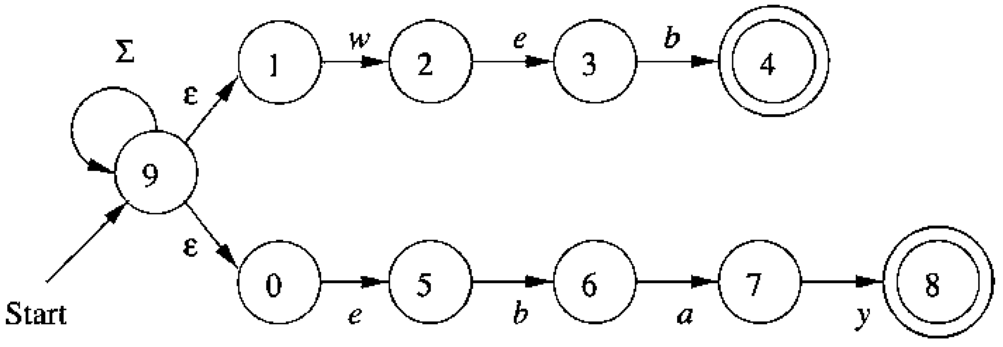


Figure 2.19: Using ϵ-transitions to help recognize keywords

2.5.2 The Formal Notation for an ϵ-NFA

We may represent an ϵ-NFA exactly as we do an NFA, with one exception: the transition function must include information about transitions on ϵ. Formally, we represent an ϵ-NFA A by $A = (Q, \Sigma, \delta, q_0, F)$, where all components have their same interpretation as for an NFA, except that δ is now a function that takes as arguments:

1. A state in Q, and

2. A member of $\Sigma \cup \{\epsilon\}$, that is, either an input symbol, or the symbol ϵ. We require that ϵ, the symbol for the empty string, cannot be a member of the alphabet Σ, so no confusion results.

Example 2.18: The ϵ-NFA of Fig. 2.18 is represented formally as

$$E = (\{q_0, q_1, \ldots, q_5\}, \{., +, -, 0, 1, \ldots, 9\}, \delta, q_0, \{q_5\})$$

where δ is defined by the transition table in Fig. 2.20. □

	ϵ	$+, -$	$.$	$0, 1, \ldots, 9$
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 2.20: Transition table for Fig. 2.18

2.5.3 Epsilon-Closures

We shall proceed to give formal definitions of an extended transition function for ϵ-NFA's, which leads to the definition of acceptance of strings and languages by these automata, and eventually lets us explain why ϵ-NFA's can be simulated by DFA's. However, we first need to learn a central definition, called the *ϵ-closure* of a state. Informally, we ϵ-close a state q by following all transitions out of q that are labeled ϵ. However, when we get to other states by following ϵ, we follow the ϵ-transitions out of those states, and so on, eventually finding every state that can be reached from q along any path whose arcs are all labeled ϵ. Formally, we define the ϵ-closure $\text{ECLOSE}(q)$ recursively, as follows:

BASIS: State q is in $\text{ECLOSE}(q)$.

INDUCTION: If state p is in $\text{ECLOSE}(q)$, and there is a transition from state p to state r labeled ϵ, then r is in $\text{ECLOSE}(q)$. More precisely, if δ is the transition function of the ϵ-NFA involved, and p is in $\text{ECLOSE}(q)$, then $\text{ECLOSE}(q)$ also contains all the states in $\delta(p, \epsilon)$.

Example 2.19: For the automaton of Fig. 2.18, each state is its own ϵ-closure, with two exceptions: $\text{ECLOSE}(q_0) = \{q_0, q_1\}$ and $\text{ECLOSE}(q_3) = \{q_3, q_5\}$. The reason is that there are only two ϵ-transitions, one that adds q_1 to $\text{ECLOSE}(q_0)$ and the other that adds q_5 to $\text{ECLOSE}(q_3)$.

A more complex example is given in Fig. 2.21. For this collection of states, which may be part of some ϵ-NFA, we can conclude that

$$\text{ECLOSE}(1) = \{1, 2, 3, 4, 6\}$$

Each of these states can be reached from state 1 along a path exclusively labeled ϵ. For example, state 6 is reached by the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$. State 7 is not in $\text{ECLOSE}(1)$, since although it is reachable from state 1, the path must use the arc $4 \rightarrow 5$ that is not labeled ϵ. The fact that state 6 is also reached from state 1 along a path $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ that has non-ϵ transitions is unimportant. The existence of one path with all labels ϵ is sufficient to show state 6 is in $\text{ECLOSE}(1)$. □

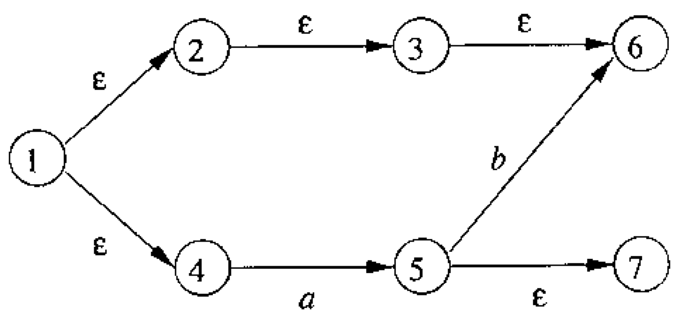


Figure 2.21: Some states and transitions

2.5.4 Extended Transitions and Languages for ϵ-NFA's

The ϵ-closure allows us to explain easily what the transitions of an ϵ-NFA look like when given a sequence of (non-ϵ) inputs. From there, we can define what it means for an ϵ-NFA to accept its input.

Suppose that $E = (Q, \Sigma, \delta, q_0, F)$ is an ϵ-NFA. We first define $\hat{\delta}$, the extended transition function, to reflect what happens on a sequence of inputs. The intent is that $\hat{\delta}(q, w)$ is the set of states that can be reached along a path whose labels, when concatenated, form the string w. As always, ϵ's along this path do not contribute to w. The appropriate recursive definition of $\hat{\delta}$ is:

BASIS: $\hat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$. That is, if the label of the path is ϵ, then we can follow only ϵ-labeled arcs extending from state q; that is exactly what ECLOSE does.

INDUCTION: Suppose w is of the form xa, where a is the last symbol of w. Note that a is a member of Σ; it cannot be ϵ, which is not in Σ. We compute $\hat{\delta}(q, w)$ as follows:

1. Let $\{p_1, p_2, \ldots, p_k\}$ be $\hat{\delta}(q, x)$. That is, the p_i's are all and only the states that we can reach from q following a path labeled x. This path may end with one or more transitions labeled ϵ, and may have other ϵ-transitions, as well.

2. Let $\bigcup_{i=1}^{k} \delta(p_i, a)$ be the set $\{r_1, r_2, \ldots, r_m\}$. That is, follow all transitions labeled a from states we can reach from q along paths labeled x. The r_j's are *some* of the states we can reach from q along paths labeled w. The additional states we can reach are found from the r_j's by following ϵ-labeled arcs in step (3), below.

3. Then $\hat{\delta}(q, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$. This additional closure step includes all the paths from q labeled w, by considering the possibility that there are additional ϵ-labeled arcs that we can follow after making a transition on the final "real" symbol, a.

Example 2.20: Let us compute $\hat{\delta}(q_0, 5.6)$ for the ϵ-NFA of Fig. 2.18. A summary of the steps needed are as follows:

- $\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$.
- Compute $\hat{\delta}(q_0, 5)$ as follows:
  1. First compute the transitions on input 5 from the states q_0 and q_1 that we obtained in the calculation of $\hat{\delta}(q_0, \epsilon)$, above. That is, we compute $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.
  2. Next, ϵ-close the members of the set computed in step (1). We get $\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$. That set is $\hat{\delta}(q_0, 5)$. This two-step pattern repeats for the next two symbols.

- Compute $\hat{\delta}(q_0, \texttt{5.})$ as follows:

  1. First compute $\delta(q_1, \texttt{.}) \cup \delta(q_4, \texttt{.}) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$.
  2. Then compute

  $$\hat{\delta}(q_0, \texttt{5.}) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$$

- Compute $\hat{\delta}(q_0, \texttt{5.6})$ as follows:

  1. First compute $\delta(q_2, \texttt{6}) \cup \delta(q_3, \texttt{6}) \cup \delta(q_5, \texttt{6}) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$.
  2. Then compute $\hat{\delta}(q_0, \texttt{5.6}) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$.

□

Now, we can define the language of an ϵ-NFA $E = (Q, \Sigma, \delta, q_0, F)$ in the expected way: $L(E) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$. That is, the language of E is the set of strings w that take the start state to at least one accepting state. For instance, we saw in Example 2.20 that $\hat{\delta}(q_0, \texttt{5.6})$ contains the accepting state q_5, so the string 5.6 is in the language of that ϵ-NFA.

2.5.5 Eliminating ϵ-Transitions

Given any ϵ-NFA E, we can find a DFA D that accepts the same language as E. The construction we use is very close to the subset construction, as the states of D are subsets of the states of E. The only difference is that we must incorporate ϵ-transitions of E, which we do through the mechanism of the ϵ-closure.

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Then the equivalent DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

is defined as follows:

1. Q_D is the set of subsets of Q_E. More precisely, we shall find that all accessible states of D are *ϵ-closed* subsets of Q_E, that is, sets $S \subseteq Q_E$ such that $S = \text{ECLOSE}(S)$. Put another way, the ϵ-closed sets of states S are those such that any ϵ-transition out of one of the states in S leads to a state that is also in S. Note that $\emptyset$ is an ϵ-closed set.

2. $q_D = \text{ECLOSE}(q_0)$; that is, we get the start state of D by closing the set consisting of only the start state of E. Note that this rule differs from the original subset construction, where the start state of the constructed automaton was just the set containing the start state of the given NFA.

3. F_D is those sets of states that contain at least one accepting state of E. That is, $F_D = \{S \mid S \text{ is in } Q_D \text{ and } S \cap F_E \neq \emptyset\}$.

4. $\delta_D(S, a)$ is computed, for all a in Σ and sets S in Q_D by:

(a) Let $S = \{p_1, p_2, \ldots, p_k\}$.

(b) Compute $\bigcup_{i=1}^{k} \delta_E(p_i, a)$; let this set be $\{r_1, r_2, \ldots, r_m\}$.

(c) Then $\delta_D(S, a) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$.

Example 2.21: Let us eliminate ϵ-transitions from the ϵ-NFA of Fig. 2.18, which we shall call E in what follows. From E, we construct an DFA D, which is shown in Fig. 2.22. However, to avoid clutter, we omitted from Fig. 2.22 the dead state $\emptyset$ and all transitions to the dead state. You should imagine that for each state shown in Fig. 2.22 there are additional transitions from any state to $\emptyset$ on any input symbols for which a transition is not indicated. Also, the state $\emptyset$ has transitions to itself on all input symbols.

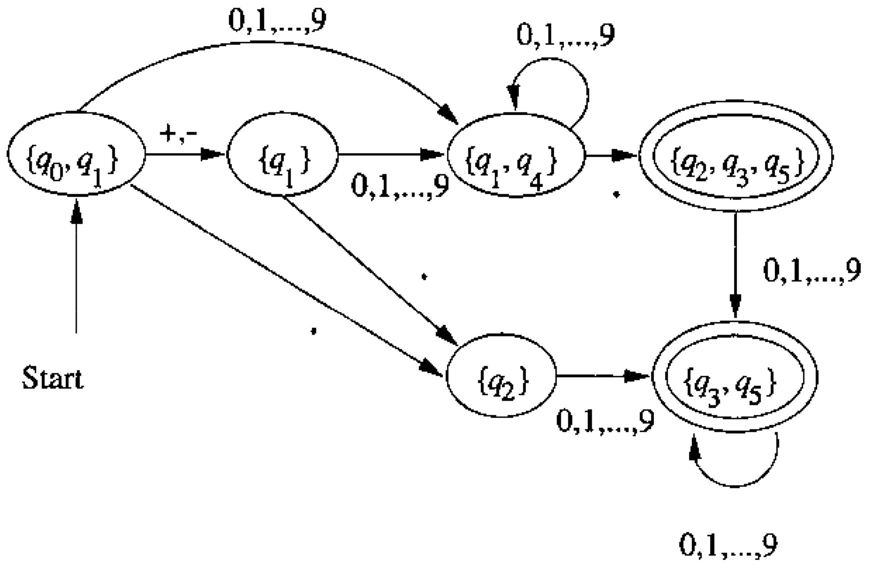


Figure 2.22: The DFA D that eliminates ϵ-transitions from Fig. 2.18

Since the start state of E is q_0, the start state of D is ECLOSE(q_0), which is $\{q_0, q_1\}$. Our first job is to find the successors of q_0 and q_1 on the various symbols in Σ; note that these symbols are the plus and minus signs, the dot, and the digits 0 through 9. On + and −, q_1 goes nowhere in Fig. 2.18, while q_0 goes to q_1. Thus, to compute $\delta_D(\{q_0, q_1\}, +)$ we start with $\{q_1\}$ and ϵ-close it. Since there are no ϵ-transitions out of q_1, we have $\delta_D(\{q_0, q_1\}, +) = \{q_1\}$. Similarly, $\delta_D(\{q_0, q_1\}, -) = \{q_1\}$. These two transitions are shown by one arc in Fig. 2.22.

Next, we need to compute $\delta_D(\{q_0, q_1\}, .)$. Since q_0 goes nowhere on the dot, and q_1 goes to q_2 in Fig. 2.18, we must ϵ-close $\{q_2\}$. As there are no ϵ-transitions out of q_2, this state is its own closure, so $\delta_D(\{q_0, q_1\}, .) = \{q_2\}$.

Finally, we must compute $\delta_D(\{q_0, q_1\}, 0)$, as an example of the transitions from $\{q_0, q_1\}$ on all the digits. We find that q_0 goes nowhere on the digits, but q_1 goes to both q_1 and q_4. Since neither of those states have ϵ-transitions out, we conclude $\delta_D(\{q_0, q_1\}, 0) = \{q_1, q_4\}$, and likewise for the other digits.

We have now explained the arcs out of $\{q_0, q_1\}$ in Fig. 2.22. The other transitions are computed similarly, and we leave them for you to check. Since q_5 is the only accepting state of E, the accepting states of D are those accessible states that contain q_5. We see these two sets $\{q_3, q_5\}$ and $\{q_2, q_3, q_5\}$ indicated by double circles in Fig. 2.22. □

Theorem 2.22: A language L is accepted by some ϵ-NFA if and only if L is accepted by some DFA.

PROOF: (If) This direction is easy. Suppose $L = L(D)$ for some DFA. Turn D into an ϵ-DFA E by adding transitions $\delta(q, \epsilon) = \emptyset$ for all states q of D. Technically, we must also convert the transitions of D on input symbols, e.g., $\delta_D(q, a) = p$ into an NFA-transition to the set containing only p, that is $\delta_E(q, a) = \{p\}$. Thus, the transitions of E and D are the same, but E explicitly states that there are no transitions out of any state on ϵ.

(Only-if) Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ϵ-NFA. Apply the modified subset construction described above to produce the DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

We need to show that $L(D) = L(E)$, and we do so by showing that the extended transition functions of E and D are the same. Formally, we show $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ by induction on the length of w.

BASIS: If $|w| = 0$, then $w = \epsilon$. We know $\hat{\delta}_E(q_0, \epsilon) = \text{ECLOSE}(q_0)$. We also know that $q_D = \text{ECLOSE}(q_0)$, because that is how the start state of D is defined. Finally, for a DFA, we know that $\hat{\delta}(p, \epsilon) = p$ for any state p, so in particular, $\hat{\delta}_D(q_D, \epsilon) = \text{ECLOSE}(q_0)$. We have thus proved that $\hat{\delta}_E(q_0, \epsilon) = \hat{\delta}_D(q_D, \epsilon)$.

INDUCTION: Suppose $w = xa$, where a is the final symbol of w, and assume that the statement holds for x. That is, $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x)$. Let both these sets of states be $\{p_1, p_2, \ldots, p_k\}$.

By the definition of $\hat{\delta}$ for ϵ-NFA's, we compute $\hat{\delta}_E(q_0, w)$ by:

1. Let $\{r_1, r_2, \ldots, r_m\}$ be $\bigcup_{i=1}^{k} \delta_E(p_i, a)$.

2. Then $\hat{\delta}_E(q_0, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$.

If we examine the construction of DFA D in the modified subset construction above, we see that $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ is constructed by the same two steps (1) and (2) above. Thus, $\hat{\delta}_D(q_D, w)$, which is $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ is the same set as $\hat{\delta}_E(q_0, w)$. We have now proved that $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ and completed the inductive part. □

2.5.6 Exercises for Section 2.5

* **Exercise 2.5.1:** Consider the following ϵ-NFA.

	ϵ	a	b	c
$\rightarrow p$	$\emptyset$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\emptyset$
$*r$	$\{q\}$	$\{r\}$	$\emptyset$	$\{p\}$

a) Compute the ϵ-closure of each state.

b) Give all the strings of length three or less accepted by the automaton.

c) Convert the automaton to a DFA.

Exercise 2.5.2: Repeat Exercise 2.5.1 for the following ϵ-NFA:

	ϵ	a	b	c
$\rightarrow p$	$\{q,r\}$	$\emptyset$	$\{q\}$	$\{r\}$
q	$\emptyset$	$\{p\}$	$\{r\}$	$\{p,q\}$
$*r$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Exercise 2.5.3: Design ϵ-NFA's for the following languages. Try to use ϵ-transitions to simplify your design.

a) The set of strings consisting of zero or more a's followed by zero or more b's, followed by zero or more c's.

! b) The set of strings that consist of either 01 repeated one or more times or 010 repeated one or more times.

! c) The set of strings of 0's and 1's such that at least one of the last ten positions is a 1.

2.6 Summary of Chapter 2

✦ *Deterministic Finite Automata*: A DFA has a finite set of states and a finite set of input symbols. One state is designated the start state, and zero or more states are accepting states. A transition function determines how the state changes each time an input symbol is processed.

✦ *Transition Diagrams*: It is convenient to represent automata by a graph in which the nodes are the states, and arcs are labeled by input symbols, indicating the transitions of that automaton. The start state is designated by an arrow, and the accepting states by double circles.

- ✦ *Language of an Automaton*: The automaton accepts strings. A string is accepted if, starting in the start state, the transitions caused by processing the symbols of that string one-at-a-time lead to an accepting state. In terms of the transition diagram, a string is accepted if it is the label of a path from the start state to some accepting state.

- ✦ *Nondeterministic Finite Automata*: The NFA differs from the DFA in that the NFA can have any number of transitions (including zero) to next states from a given state on a given input symbol.

- ✦ *The Subset Construction*: By treating sets of states of an NFA as states of a DFA, it is possible to convert any NFA to a DFA that accepts the same language.

- ✦ *ϵ-Transitions*: We can extend the NFA by allowing transitions on an empty input, i.e., no input symbol at all. These extended NFA's can be converted to DFA's accepting the same language.

- ✦ *Text-Searching Applications*: Nondeterministic finite automata are a useful way to represent a pattern matcher that scans a large body of text for one or more keywords. These automata are either simulated directly in software or are first converted to a DFA, which is then simulated.

2.7 References for Chapter 2

The formal study of finite-state systems is generally regarded as originating with [2]. However, this work was based on a "neural nets" model of computing, rather than the finite automaton we know today. The conventional DFA was independently proposed, in several similar variations, by [1], [3], and [4]. The nondeterministic finite automaton and the subset construction are from [5].

1. D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.* **257**:3-4 (1954), pp. 161–190 and 275–303.

2. W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervious activity," *Bull. Math. Biophysics* **5** (1943), pp. 115–133.

3. G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal* **34**:5 (1955), pp. 1045–1079.

4. E. F. Moore, "Gedanken experiments on sequential machines," in [6], pp. 129–153.

5. M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM J. Research and Development* **3**:2 (1959), pp. 115–125.

6. C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956.

Chapter 3

Regular Expressions and Languages

We begin this chapter by introducing the notation called "regular expressions." These expressions are another type of language-defining notation, which we sampled briefly in Section 1.1.2. Regular expressions also may be thought of as a "programming language," in which we express some important applications, such as text-search applications or compiler components. Regular expressions are closely related to nondeterministic finite automata and can be thought of as a "user-friendly" alternative to the NFA notation for describing software components.

In this chapter, after defining regular expressions, we show that they are capable of defining all and only the regular languages. We discuss the way that regular expressions are used in several software systems. Then, we examine the algebraic laws that apply to regular expressions. They have significant resemblance to the algebraic laws of arithmetic, yet there are also some important differences between the algebras of regular expressions and arithmetic expressions.

3.1 Regular Expressions

Now, we switch our attention from machine-like descriptions of languages — deterministic and nondeterministic finite automata — to an algebraic description: the "regular expression." We shall find that regular expressions can define exactly the same languages that the various forms of automata describe: the regular languages. However, regular expressions offer something that automata do not: a declarative way to express the strings we want to accept. Thus, regular expressions serve as the input language for many systems that process strings. Examples include:

1. Search commands such as the UNIX **grep** or equivalent commands for finding strings that one sees in Web browsers or text-formatting systems. These systems use a regular-expression-like notation for describing patterns that the user wants to find in a file. Different search systems convert the regular expression into either a DFA or an NFA, and simulate that automaton on the file being searched.

2. Lexical-analyzer generators, such as Lex or Flex. Recall that a lexical analyzer is the component of a compiler that breaks the source program into logical units (called *tokens*) of one or more characters that have a shared significance. Examples of tokens include keywords (e.g., `while`), identifiers (e.g., any letter followed by zero or more letters and/or digits), and signs, such as + or <=. A lexical-analyzer generator accepts descriptions of the forms of tokens, which are essentially regular expressions, and produces a DFA that recognizes which token appears next on the input.

3.1.1 The Operators of Regular Expressions

Regular expressions denote languages. For a simple example, the regular expression $\mathbf{01^* + 10^*}$ denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by any number of 0's. We do not expect you to know at this point how to interpret regular expressions, so our statement about the language of this expression must be accepted on faith for the moment. We shortly shall define all the symbols used in this expression, so you can see why our interpretation of this regular expression is the correct one. Before describing the regular-expression notation, we need to learn the three operations on languages that the operators of regular expressions represent. These operations are:

1. The *union* of two languages L and M, denoted $L \cup M$, is the set of strings that are in either L or M, or both. For example, if $L = \{001, 10, 111\}$ and $M = \{\epsilon, 001\}$, then $L \cup M = \{\epsilon, 10, 001, 111\}$.

2. The *concatenation* of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. Recall Section 1.5.2, where we defined the concatenation of a pair of strings; one string is followed by the other to form the result of the concatenation. We denote concatenation of languages either with a dot or with no operator at all, although the concatenation operator is frequently called "dot." For example, if $L = \{001, 10, 111\}$ and $M = \{\epsilon, 001\}$, then $L.M$, or just LM, is $\{001, 10, 111, 001001, 10001, 111001\}$. The first three strings in LM are the strings in L concatenated with ϵ. Since ϵ is the identity for concatenation, the resulting strings are the same as the strings of L. However, the last three strings in LM are formed by taking each string in L and concatenating it with the second string in M, which is 001. For instance, 10 from L concatenated with 001 from M gives us 10001 for LM.

3. The *closure* (or *star*, or *Kleene closure*)[1] of a language L is denoted L^* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them. For instance, if $L = \{0, 1\}$, then L^* is all strings of 0's and 1's. If $L = \{0, 11\}$, then L^* consists of those strings of 0's and 1's such that the 1's come in pairs, e.g., 011, 11110, and ϵ, but not 01011 or 101. More formally, L^* is the infinite union $\cup_{i\geq 0} L^i$, where $L^0 = \{\epsilon\}$, $L^1 = L$, and L^i, for $i > 1$ is $LL \cdots L$ (the concatenation of i copies of L).

Example 3.1 : Since the idea of the closure of a language is somewhat tricky, let us study a few examples. First, let $L = \{0, 11\}$. $L^0 = \{\epsilon\}$, independent of what language L is; the 0th power represents the selection of zero strings from L. $L^1 = L$, which represents the choice of one string from L. Thus, the first two terms in the expansion of L^* give us $\{\epsilon, 0, 11\}$.

Next, consider L^2. We pick two strings from L, with repetitions allowed, so there are four choices. These four selections give us $L^2 = \{00, 011, 110, 1111\}$. Similarly, L^3 is the set of strings that may be formed by making three choices of the two strings in L and gives us

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

To compute L^*, we must compute L^i for each i, and take the union of all these languages. L^i has 2^i members. Although each L^i is finite, the union of the infinite number of terms L^i is generally an infinite language, as it is in our example.

Now, let L be the set of all strings of 0's. Note that L is infinite, unlike our previous example, which is a finite language. However, it is not hard to discover what L^* is. $L^0 = \{\epsilon\}$, as always. $L^1 = L$. L^2 is the set of strings that can be formed by taking one string of 0's and concatenating it with another string of 0's. The result is still a string of 0's. In fact, every string of 0's can be written as the concatenation of two strings of 0's (don't forget that ϵ is a "string of 0's"; this string can always be one of the two strings that we concatenate). Thus, $L^2 = L$. Likewise, $L^3 = L$, and so on. Thus, the infinite union $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$ is L in the particular case that the language L the set of all strings of 0's.

For a final example, $\emptyset^* = \{\epsilon\}$. Note that $\emptyset^0 = \{\epsilon\}$, while $\emptyset^i$, for any $i \geq 1$, is empty, since we can't select any strings from the empty set. In fact, $\emptyset$ is one of only two languages whose closure is *not* infinite. □

3.1.2 Building Regular Expressions

Algebras of all kinds start with some elementary expressions, usually constants and/or variables. Algebras then allow us to construct more expressions by

[1]The term "Kleene closure" refers to S. C. Kleene, who originated the regular expression notation and this operator.

Use of the Star Operator

We saw the star operator first in Section 1.5.2, where we applied it to an alphabet, e.g., Σ^*. That operator formed all strings whose symbols were chosen from alphabet Σ. The closure operator is essentially the same, although there is a subtle distinction of types.

Suppose L is the language containing strings of length 1, and for each symbol a in Σ there is a string a in L. Then, although L and Σ "look" the same, they are of different types; L is a set of strings, and Σ is a set of symbols. On the other hand, L^* denotes the same language as Σ^*.

applying a certain set of operators to these elementary expressions and to previously constructed expressions. Usually, some method of grouping operators with their operands, such as parentheses, is required as well. For instance, the familiar arithmetic algebra starts with constants such as integers and real numbers, plus variables, and builds more complex expressions with arithmetic operators such as $+$ and $\times$.

The algebra of regular expressions follows this pattern, using constants and variables that denote languages, and operators for the three operations of Section 3.1.1 —union, dot, and star. We can describe the regular expressions recursively, as follows. In this definition, we not only describe what the legal regular expressions are, but for each regular expression E, we describe the language it represents, which we denote $L(E)$.

BASIS: The basis consists of three parts:

1. The constants ϵ and $\emptyset$ are regular expressions, denoting the languages $\{\epsilon\}$ and $\emptyset$, respectively. That is, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \emptyset$.

2. If a is any symbol, then $\mathbf{a}$ is a regular expression. This expression denotes the language $\{a\}$. That is, $L(\mathbf{a}) = \{a\}$. Note that we use boldface font to denote an expression corresponding to a symbol. The correspondence, e.g. that $\mathbf{a}$ refers to a, should be obvious.

3. A variable, usually capitalized and italic such as L, is a variable, representing any language.

INDUCTION: There are four parts to the inductive step, one for each of the three operators and one for the introduction of parentheses.

1. If E and F are regular expressions, then $E + F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E+F) = L(E) \cup L(F)$.

2. If E and F are regular expressions, then EF is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. That is, $L(EF) = L(E)L(F)$.

Expressions and Their Languages

Strictly speaking, a regular expression E is just an expression, not a language. We should use $L(E)$ when we want to refer to the language that E denotes. However, it is common usage to refer to say "E" when we really mean "$L(E)$." We shall use this convention as long as it is clear we are talking about a language and not about a regular expression.

Note that the dot can optionally be used to denote the concatenation operator, either as an operation on languages or as the operator in a regular expression. For instance, $\mathbf{0.1}$ is a regular expression meaning the same as $\mathbf{01}$ and representing the language $\{01\}$. However, we shall avoid the dot as concatenation in regular expressions.[2]

3. If E is a regular expression, then E^* is a regular expression, denoting the closure of $L(E)$. That is, $L(E^*) = \big(L(E)\big)^*$.

4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally; $L\big((E)\big) = L(E)$.

Example 3.2: Let us write a regular expression for the set of strings that consist of alternating 0's and 1's. First, let us develop a regular expression for the language consisting of the single string 01. We can then use the star operator to get an expression for all strings of the form $0101\cdots01$.

The basis rule for regular expressions tells us that $\mathbf{0}$ and $\mathbf{1}$ are expressions denoting the languages $\{0\}$ and $\{1\}$, respectively. If we concatenate the two expressions, we get a regular expression for the language $\{01\}$; this expression is $\mathbf{01}$. As a general rule, if we want a regular expression for the language consisting of only the string w, we use w itself as the regular expression. Note that in the regular expression, the symbols of w will normally be written in boldface, but the change of font is only to help you distinguish expressions from strings and should not be taken as significant.

Now, to get all strings consisting of zero or more occurrences of 01, we use the regular expression $(\mathbf{01})^*$. Note that we first put parentheses around $\mathbf{01}$, to avoid confusing with the expression $\mathbf{01}^*$, whose language is all strings consisting of a 0 and any number of 1's. The reason for this interpretation is explained in Section 3.1.3, but briefly, star takes precedence over dot, and therefore the argument of the star is selected before performing any concatenations.

However, $L\big((\mathbf{01})^*\big)$ is not exactly the language that we want. It includes only those strings of alternating 0's and 1's that begin with 0 and end with 1. We also need to consider the possibility that there is a 1 at the beginning and/or

[2] In fact, UNIX regular expressions use the dot for an entirely different purpose: representing any ASCII character.

a 0 at the end. One approach is to construct three more regular expressions that handle the other three possibilities. That is, $(\mathbf{10})^*$ represents those alternating strings that begin with 1 and end with 0, while $\mathbf{0}(\mathbf{10})^*$ can be used for strings that both begin and end with 0 and $\mathbf{1}(\mathbf{01})^*$ serves for strings that begin and end with 1. The entire regular expression is

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Notice that we use the $+$ operator to take the union of the four languages that together give us all the strings with alternating 0's and 1's.

However, there is another approach that yields a regular expression that looks rather different and is also somewhat more succinct. Start again with the expression $(\mathbf{01})^*$. We can add an optional 1 at the beginning if we concatenate on the left with the expression $\epsilon + \mathbf{1}$. Likewise, we add an optional 0 at the end with the expression $\epsilon + \mathbf{0}$. For instance, using the definition of the $+$ operator:

$$L(\epsilon + \mathbf{1}) = L(\epsilon) \cup L(\mathbf{1}) = \{\epsilon\} \cup \{1\} = \{\epsilon, 1\}$$

If we concatenate this language with any other language L, the ϵ choice gives us all the strings in L, while the 1 choice gives us $1w$ for every string w in L. Thus, another expression for the set of strings that alternate 0's and 1's is:

$$(\epsilon + \mathbf{1})(\mathbf{01})^*(\epsilon + \mathbf{0})$$

Note that we need parentheses around each of the added expressions, to make sure the operators group properly. □

3.1.3 Precedence of Regular-Expression Operators

Like other algebras, the regular-expression operators have an assumed order of "precedence," which means that operators are associated with their operands in a particular order. We are familiar with the notion of precedence from ordinary arithmetic expressions. For instance, we know that $xy+z$ groups the product xy before the sum, so it is equivalent to the parenthesized expression $(xy) + z$ and not to the expression $x(y + z)$. Similarly, we group two of the same operators from the left in arithmetic, so $x - y - z$ is equivalent to $(x - y) - z$, and not to $x - (y - z)$. For regular expressions, the following is the order of precedence for the operators:

1. The star operator is of highest precedence. That is, it applies only to the smallest sequence of symbols to its left that is a well-formed regular expression.

2. Next in precedence comes the concatenation or "dot" operator. After grouping all stars to their operands, we group concatenation operators to their operands. That is, all expressions that are *juxtaposed* (adjacent, with no intervening operator) are grouped together. Since concatenation

is an associative operator it does not matter in what order we group consecutive concatenations, although if there is a choice to be made, you should group them from the left. For instance, **012** is grouped **(01)2**.

3. Finally, all unions (+ operators) are grouped with their operands. Since union is also associative, it again matters little in which order consecutive unions are grouped, but we shall assume grouping from the left.

Of course, sometimes we do not want the grouping in a regular expression to be as required by the precedence of the operators. If so, we are free to use parentheses to group operands exactly as we choose. In addition, there is never anything wrong with putting parentheses around operands that you want to group, even if the desired grouping is implied by the rules of precedence.

Example 3.3: The expression $\mathbf{01^* + 1}$ is grouped $\big(\mathbf{0}(\mathbf{1}^*)\big) + \mathbf{1}$. The star operator is grouped first. Since the symbol **1** immediately to its left is a legal regular expression, that alone is the operand of the star. Next, we group the concatenation between **0** and $(\mathbf{1}^*)$, giving us the expression $\big(\mathbf{0}(\mathbf{1}^*)\big)$. Finally, the union operator connects the latter expression and the expression to its right, which is **1**.

Notice that the language of the given expression, grouped according to the precedence rules, is the string 1 plus all strings consisting of a 0 followed by any number of 1's (including none). Had we chosen to group the dot before the star, we could have used parentheses, as $(\mathbf{01})^* + \mathbf{1}$. The language of this expression is the string 1 and all strings that repeat 01, zero or more times. Had we wished to group the union first, we could have added parentheses around the union to make the expression $\mathbf{0}(\mathbf{1}^* + \mathbf{1})$. That expression's language is the set of strings that begin with 0 and have any number of 1's following. □

3.1.4 Exercises for Section 3.1

Exercise 3.1.1: Write regular expressions for the following languages:

* a) The set of strings over alphabet $\{a, b, c\}$ containing at least one a and at least one b.

b) The set of strings of 0's and 1's whose tenth symbol from the right end is 1.

c) The set of strings of 0's and 1's with at most one pair of consecutive 1's.

! **Exercise 3.1.2:** Write regular expressions for the following languages:

* a) The set of all strings of 0's and 1's such that every pair of adjacent 0's appears before any pair of adjacent 1's.

b) The set of strings of 0's and 1's whose number of 0's is divisible by five.

!! **Exercise 3.1.3:** Write regular expressions for the following languages:

a) The set of all strings of 0's and 1's not containing 101 as a substring.

b) The set of all strings with an equal number of 0's and 1's, such that no prefix has two more 0's than 1's, nor two more 1's than 0's.

c) The set of strings of 0's and 1's whose number of 0's is divisible by five and whose number of 1's is even.

! **Exercise 3.1.4:** Give English descriptions of the languages of the following regular expressions:

* a) $(\mathbf{1}+\epsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$.

b) $(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0}+\mathbf{1})^*$.

c) $(\mathbf{0}+\mathbf{10})^*\mathbf{1}^*$.

*! **Exercise 3.1.5:** In Example 3.1 we pointed out that $\emptyset$ is one of two languages whose closure is finite. What is the other?

3.2 Finite Automata and Regular Expressions

While the regular-expression approach to describing languages is fundamentally different from the finite-automaton approach, these two notations turn out to represent exactly the same set of languages, which we have termed the "regular languages." We have already shown that deterministic finite automata, and the two kinds of nondeterministic finite automata — with and without ϵ-transitions — accept the same class of languages. In order to show that the regular expressions define the same class, we must show that:

1. Every language defined by one of these automata is also defined by a regular expression. For this proof, we can assume the language is accepted by some DFA.

2. Every language defined by a regular expression is defined by one of these automata. For this part of the proof, the easiest is to show that there is an NFA with ϵ-transitions accepting the same language.

Figure 3.1 shows all the equivalences we have proved or will prove. An arc from class X to class Y means that we prove every language defined by class X is also defined by class Y. Since the graph is strongly connected (i.e., we can get from each of the four nodes to any other node) we see that all four classes are really the same.

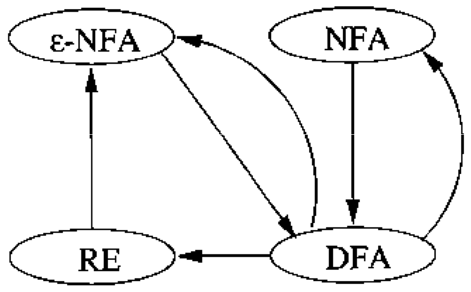


Figure 3.1: Plan for showing the equivalence of four different notations for regular languages

3.2.1 From DFA's to Regular Expressions

The construction of a regular expression to define the language of any DFA is surprisingly tricky. Roughly, we build expressions that describe sets of strings that label certain paths in the DFA's transition diagram. However, the paths are allowed to pass through only a limited subset of the states. In an inductive definition of these expressions, we start with the simplest expressions that describe paths that are not allowed to pass through *any* states (i.e., they are single nodes or single arcs), and inductively build the expressions that let the paths go through progressively larger sets of states. Finally, the paths are allowed to go through any state; i.e., the expressions we generate at the end represent all possible paths. These ideas appear in the proof of the following theorem.

Theorem 3.4: If $L = L(A)$ for some DFA A, then there is a regular expression R such that $L = L(R)$.

PROOF: Let us suppose that A's states are $\{1, 2, \ldots, n\}$ for some integer n. No matter what the states of A actually are, there will be n of them for some finite n, and by renaming the states, we can refer to the states in this manner, as if they were the first n positive integers. Our first, and most difficult, task is to construct a collection of regular expressions that describe progressively broader sets of paths in the transition diagram of A.

Let us use $R_{ij}^{(k)}$ as the name of a regular expression whose language is the set of strings w such that w is the label of a path from state i to state j in A, and that path has no intermediate node whose number is greater than k. Note that the beginning and end points of the path are not "intermediate," so there is no constraint that i and/or j be less than or equal to k.

Figure 3.2 suggests the requirement on the paths represented by $R_{ij}^{(k)}$. There, the vertical dimension represents the state, from 1 at the bottom to n at the top, and the horizontal dimension represents travel along the path. Notice that in this diagram we have shown both i and j to be greater than k, but either or both could be k or less. Also notice that the path passes through node k twice, but never goes through a state higher than k, except at the endpoints.

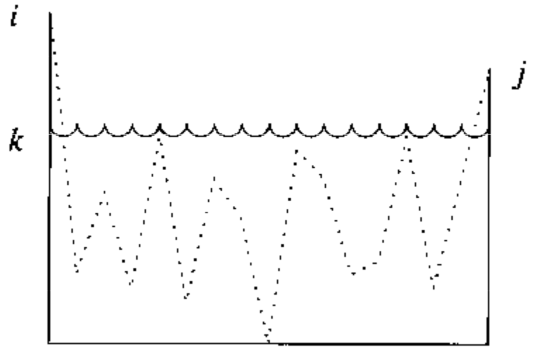


Figure 3.2: A path whose label is in the language of regular expression $R_{ij}^{(k)}$

To construct the expressions $R_{ij}^{(k)}$, we use the following inductive definition, starting at $k = 0$ and finally reaching $k = n$. Notice that when $k = n$, there is no restriction at all on the paths represented, since there *are* no states greater than n.

BASIS: The basis is $k = 0$. Since all states are numbered 1 or above, the restriction on paths is that the path must have no intermediate states at all. There are only two kinds of paths that meet such a condition:

1. An arc from node (state) i to node j.

2. A path of length 0 that consists of only some node i.

If $i \neq j$, then only case (1) is possible. We must examine the DFA A and find those input symbols a such that there is a transition from state i to state j on symbol a.

a) If there is no such symbol a, then $R_{ij}^{(0)} = \emptyset$.

b) If there is exactly one such symbol a, then $R_{ij}^{(0)} = \mathbf{a}$.

c) If there are symbols $a_1, a_2, \ldots, a_k$ that label arcs from state i to state j, then $R_{ij}^{(0)} = \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

However, if $i = j$, then the legal paths are the path of length 0 and all loops from i to itself. The path of length 0 is represented by the regular expression ϵ, since that path has no symbols along it. Thus, we add ϵ to the various expressions devised in (a) through (c) above. That is, in case (a) [no symbol a] the expression becomes ϵ, in case (b) [one symbol a] the expression becomes $\epsilon+\mathbf{a}$, and in case (c) [multiple symbols] the expression becomes $\epsilon + \mathbf{a}_1 + \mathbf{a}_2 + \cdots + \mathbf{a}_k$.

INDUCTION: Suppose there is a path from state i to state j that goes through no state higher than k. There are two possible cases to consider:

1. The path does not go through state k at all. In this case, the label of the path is in the language of $R_{ij}^{(k-1)}$.

2. The path goes through state k at least once. Then we can break the path into several pieces, as suggested by Fig. 3.3. The first goes from state i to state k without passing through k, the last piece goes from k to j without passing through k, and all the pieces in the middle go from k to itself, without passing through k. Note that if the path goes through state k only once, then there are no "middle" pieces, just a path from i to k and a path from k to j. The set of labels for all paths of this type is represented by the regular expression $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^*R_{kj}^{(k-1)}$. That is, the first expression represents the part of the path that gets to state k the first time, the second represents the portion that goes from k to itself, zero times, once, or more than once, and the third expression represents the part of the path that leaves k for the last time and goes to state j.

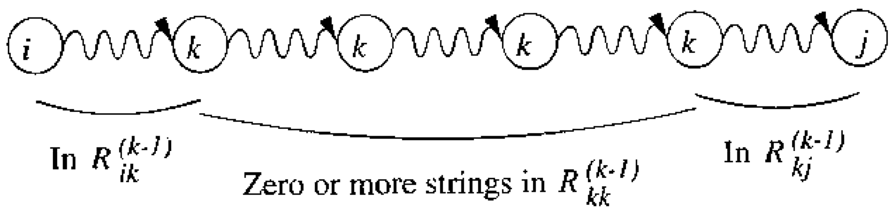


Figure 3.3: A path from i to j can be broken into segments at each point where it goes through state k

When we combine the expressions for the paths of the two types above, we have the expression

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^*R_{kj}^{(k-1)}$$

for the labels of all paths from state i to state j that go through no state higher than k. If we construct these expressions in order of increasing superscript, then since each $R_{ij}^{(k)}$ depends only on expressions with a smaller superscript, then all expressions are available when we need them.

Eventually, we have $R_{ij}^{(n)}$ for all i and j. We may assume that state 1 is the start state, although the accepting states could be any set of the states. The regular expression for the language of the automaton is then the sum (union) of all expressions $R_{1j}^{(n)}$ such that state j is an accepting state. □

Example 3.5: Let us convert the DFA of Fig. 3.4 to a regular expression. This DFA accepts all strings that have at least one 0 in them. To see why, note that the automaton goes from the start state 1 to accepting state 2 as soon as it sees an input 0. The automaton then stays in state 2 on all input sequences.

Below are the basis expressions in the construction of Theorem 3.4.

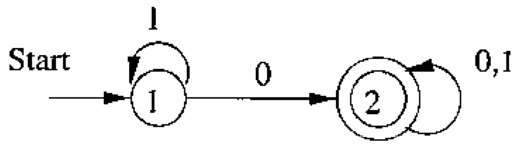


Figure 3.4: A DFA accepting all strings that have at least one 0

$R_{11}^{(0)}$	$\epsilon + \mathbf{1}$
$R_{12}^{(0)}$	$\mathbf{0}$
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\epsilon + \mathbf{0} + \mathbf{1})$

For instance, $R_{11}^{(0)}$ has the term ϵ because the beginning and ending states are the same, state 1. It has the term $\mathbf{1}$ because there is an arc from state 1 to state 1 on input 1. As another example, $R_{12}^{(0)}$ is $\mathbf{0}$ because there is an arc labeled 0 from state 1 to state 2. There is no ϵ term because the beginning and ending states are different. For a third example, $R_{21}^{(0)} = \emptyset$, because there is no arc from state 2 to state 1.

Now, we must do the induction part, building more complex expressions that first take into account paths that go through state 1, and then paths that can go through states 1 and 2, i.e., any path. The rule for computing the expressions $R_{ij}^{(1)}$ are instances of the general rule given in the inductive part of Theorem 3.4:

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

The table in Fig. 3.5 gives first the expressions computed by direct substitution into the above formula, and then a simplified expression that we can show, by ad-hoc reasoning, to represent the same language as the more complex expression.

	By direct substitution	Simplified
$R_{11}^{(1)}$	$\epsilon + \mathbf{1} + (\epsilon + \mathbf{1})(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$	$\mathbf{1}^*$
$R_{12}^{(1)}$	$\mathbf{0} + (\epsilon + \mathbf{1})(\epsilon + \mathbf{1})^*\mathbf{0}$	$\mathbf{1}^*\mathbf{0}$
$R_{21}^{(1)}$	$\emptyset + \emptyset(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon + \mathbf{0} + \mathbf{1} + \emptyset(\epsilon + \mathbf{1})^*\mathbf{0}$	$\epsilon + \mathbf{0} + \mathbf{1}$

Figure 3.5: Regular expressions for paths that can go through only state 1

For example, consider $R_{12}^{(1)}$. Its expression is $R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}$, which we get from (3.1) by substituting $i = 1$ and $j = 2$.

To understand the simplification, note the general principle that if R is any regular expression, then $(\epsilon + R)^* = R^*$. The justification is that both sides of the equation describe the language consisting of any concatenation of zero or more strings from $L(R)$. In our case, we have $(\epsilon + \mathbf{1})^* = \mathbf{1}^*$; notice that both expressions denote any number of 1's. Further, $(\epsilon+\mathbf{1})\mathbf{1}^* = \mathbf{1}^*$. Again, it can be observed that both expressions denote "any number of 1's." Thus, the original expression $R_{12}^{(1)}$ is equivalent to $\mathbf{0} + \mathbf{1}^*\mathbf{0}$. This expression denotes the language containing the string 0 and all strings consisting of a 0 preceded by any number of 1's. This language is also expressed by the simpler expression $\mathbf{1}^*\mathbf{0}$.

The simplification of $R_{11}^{(1)}$ is similar to the simplification of $R_{12}^{(1)}$ that we just considered. The simplification of $R_{21}^{(1)}$ and $R_{22}^{(1)}$ depends on two rules about how $\emptyset$ operates. For any regular expression R:

1. $\emptyset R = R\emptyset = \emptyset$. That is, $\emptyset$ is an *annihilator* for concatenation; it results in itself when concatenated, either on the left or right, with any expression. This rule makes sense, because for a string to be in the result of a concatenation, we must find strings from both arguments of the concatenation. Whenever one of the arguments is $\emptyset$, it will be impossible to find a string from that argument.

2. $\emptyset + R = R + \emptyset = R$. That is, $\emptyset$ is the identity for union; it results in the other expression whenever it appears in a union.

As a result, an expression like $\emptyset(\epsilon + \mathbf{1})^*(\epsilon + \mathbf{1})$ can be replaced by $\emptyset$. The last two simplifications should now be clear.

Now, let us compute the expressions $R_{ij}^{(2)}$. The inductive rule applied with $k = 2$ gives us:

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}(R_{22}^{(1)})^* R_{2j}^{(1)} \tag{3.2}$$

If we substitute the simplified expressions from Fig. 3.5 into (3.2), we get the expressions of Fig. 3.6. That figure also shows simplifications following the same principles that we described for Fig. 3.5.

	By direct substitution	Simplified
$R_{11}^{(2)}$	$\mathbf{1}^* + \mathbf{1}^*\mathbf{0}(\epsilon + \mathbf{0} + \mathbf{1})^*\emptyset$	$\mathbf{1}^*$
$R_{12}^{(2)}$	$\mathbf{1}^*\mathbf{0} + \mathbf{1}^*\mathbf{0}(\epsilon + \mathbf{0} + \mathbf{1})^*(\epsilon + \mathbf{0} + \mathbf{1})$	$\mathbf{1}^*\mathbf{0}(\mathbf{0} + \mathbf{1})^*$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + \mathbf{0} + \mathbf{1})(\epsilon + \mathbf{0} + \mathbf{1})^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\epsilon + \mathbf{0} + \mathbf{1} + (\epsilon + \mathbf{0} + \mathbf{1})(\epsilon + \mathbf{0} + \mathbf{1})^*(\epsilon + \mathbf{0} + \mathbf{1})$	$(\mathbf{0} + \mathbf{1})^*$

Figure 3.6: Regular expressions for paths that can go through any state

The final regular expression equivalent to the automaton of Fig. 3.4 is constructed by taking the union of all the expressions where the first state is the start state and the second state is accepting. In this example, with 1 as the start state and 2 as the only accepting state, we need only the expression $R_{12}^{(2)}$.

This expression is $\mathbf{1^*0(0+1)^*}$. It is simple to interpret this expression. Its language consists of all strings that begin with zero or more 1's, then have a 0, and then any string of 0's and 1's. Put another way, the language is all strings of 0's and 1's with at least one 0. □

3.2.2 Converting DFA's to Regular Expressions by Eliminating States

The method of Section 3.2.1 for converting a DFA to a regular expression always works. In fact, as you may have noticed, it doesn't really depend on the automaton being deterministic, and could just as well have been applied to an NFA or even an ϵ-NFA. However, the construction of the regular expression is expensive. Not only do we have to construct about n^3 expressions for an n-state automaton, but the length of the expression can grow by a factor of 4 on the average, with each of the n inductive steps, if there is no simplification of the expressions. Thus, the expressions themselves could reach on the order of 4^n symbols.

There is a similar approach that avoids duplicating work at some points. For example, all of the expressions with superscript $(k+1)$ in the construction of Theorem 3.4 use the same subexpression $(R_{kk}^{(k)})^*$; the work of writing that expression is therefore repeated n^2 times.

The approach to constructing regular expressions that we shall now learn involves eliminating states. When we eliminate a state s, all the paths that went through s no longer exist in the automaton. If the language of the automaton is not to change, we must include, on an arc that goes directly from q to p, the labels of paths that went from some state q to state p, through s. Since the label of this arc may now involve strings, rather than single symbols, and there may even be an infinite number of such strings, we cannot simply list the strings as a label. Fortunately, there is a simple, finite way to represent all such strings: use a regular expression.

Thus, we are led to consider automata that have regular expressions as labels. The language of the automaton is the union over all paths from the start state to an accepting state of the language formed by concatenating the languages of the regular expressions along that path. Note that this rule is consistent with the definition of the language for any of the varieties of automata we have considered so far. Each symbol a, or ϵ if it is allowed, can be thought of as a regular expression whose language is a single string, either $\{a\}$ or $\{\epsilon\}$. We may regard this observation as the basis of a state-elimination procedure, which we describe next.

Figure 3.7 shows a generic state s about to be eliminated. We suppose that the automaton of which s is a state has predecessor states $q_1, q_2, \ldots, q_k$ for s and successor states $p_1, p_2, \ldots, p_m$ for s. It is possible that some of the q's are also p's, but we assume that s is not among the q's or p's, even if there is a loop from s to itself, as suggested by Fig. 3.7. We also show a regular expression on each arc from one of the q's to s; expression Q_i labels the arc from q_i. Likewise,

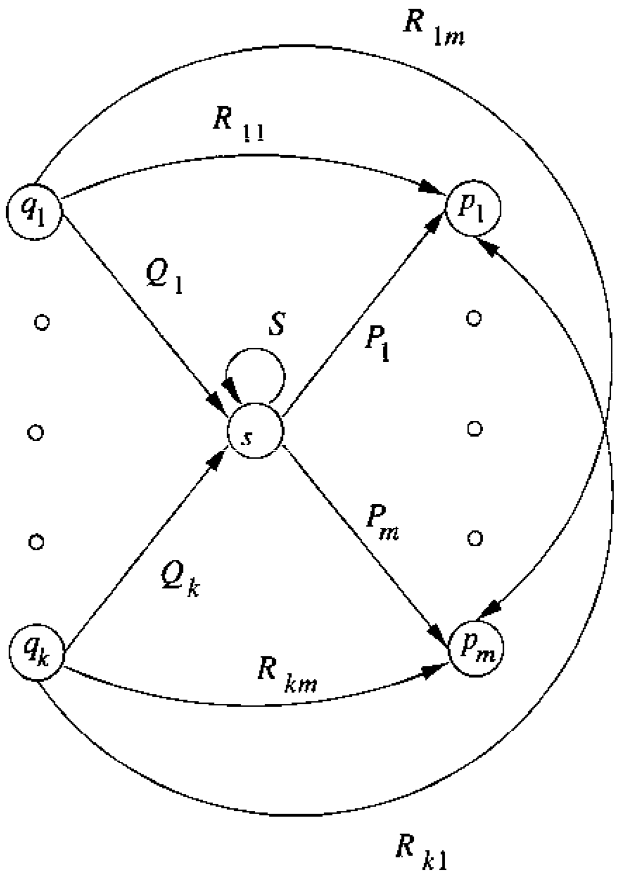


Figure 3.7: A state s about to be eliminated

we show a regular expression P_j labeling the arc from s to p_i, for all i. We show a loop on s with label S. Finally, there is a regular expression R_{ij} on the arc from q_i to p_j, for all i and j. Note that some of these arcs may not exist in the automaton, in which case we take the expression on that arc to be $\emptyset$.

Figure 3.8 shows what happens when we eliminate state s. All arcs involving state s are deleted. To compensate, we introduce, for each predecessor q_i of s and each successor p_j of s, a regular expression that represents all the paths that start at q_i, go to s, perhaps loop around s zero or more times, and finally go to p_j. The expression for these paths is $Q_iS^*P_j$. This expression is added (with the union operator) to the arc from q_i to p_j. If there was no arc $q_i \rightarrow p_j$, then first introduce one with regular expression $\emptyset$.

The strategy for constructing a regular expression from a finite automaton is as follows:

1. For each accepting state q, apply the above reduction process to produce an equivalent automaton with regular-expression labels on the arcs. Eliminate all states except q and the start state q_0.

2. If $q \neq q_0$, then we shall be left with a two-state automaton that looks like

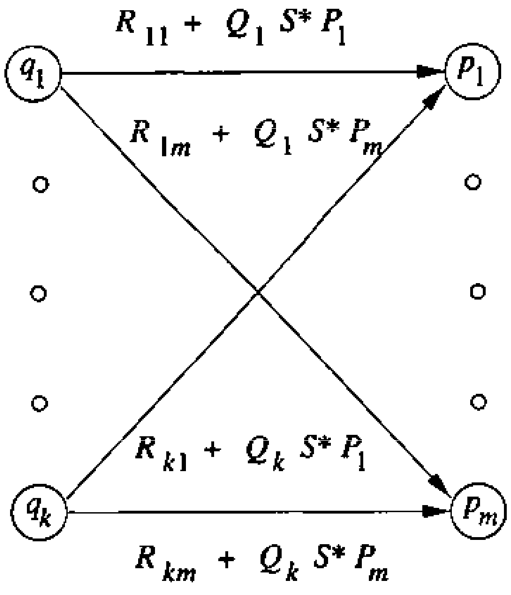


Figure 3.8: Result of eliminating state s from Fig. 3.7

Fig. 3.9. The regular expression for the accepted strings can be described in various ways. One is $(R + SU^*T)^*SU^*$. In explanation, we can go from the start state to itself any number of times, by following a sequence of paths whose labels are in either $L(R)$ or $L(SU^*T)$. The expression SU^*T represents paths that go to the accepting state via a path in $L(S)$, perhaps return to the accepting state several times using a sequence of paths with labels in $L(U)$, and then return to the start state with a path whose label is in $L(T)$. Then we must go to the accepting state, never to return to the start state, by following a path with a label in $L(S)$. Once in the accepting state, we can return to it as many times as we like, by following a path whose label is in $L(U)$.

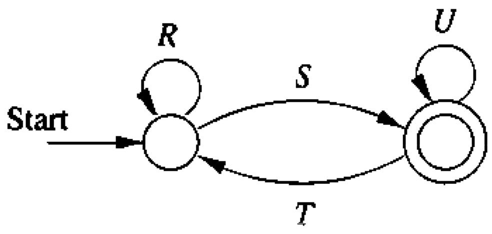


Figure 3.9: A generic two-state automaton

3. If the start state is also an accepting state, then we must also perform a state-elimination from the original automaton that gets rid of every state but the start state. When we do so, we are left with a one-state automaton that looks like Fig. 3.10. The regular expression denoting the

strings that it accepts is R^*.

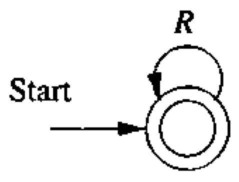


Figure 3.10: A generic one-state automaton

4. The desired regular expression is the sum (union) of all the expressions derived from the reduced automata for each accepting state, by rules (2) and (3).

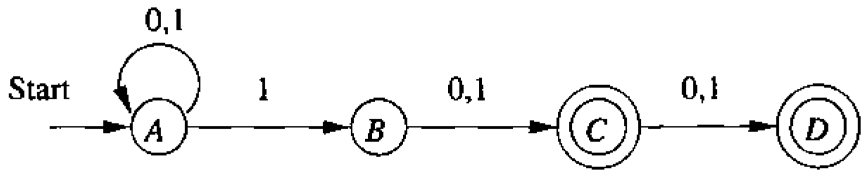


Figure 3.11: An NFA accepting strings that have a 1 either two or three positions from the end

Example 3.6: Let us consider the NFA in Fig. 3.11 that accepts all strings of 0's and 1's such that either the second or third position from the end has a 1. Our first step is to convert it to an automaton with regular expression labels. Since no state elimination has been performed, all we have to do is replace the labels "0,1" with the equivalent regular expression $\mathbf{0} + \mathbf{1}$. The result is shown in Fig. 3.12.

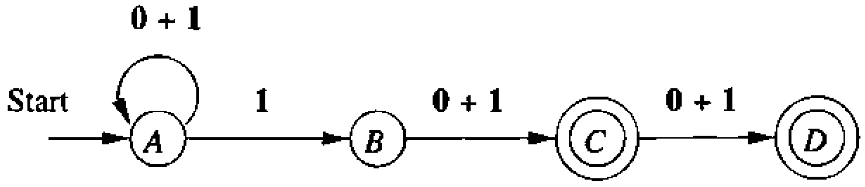


Figure 3.12: The automaton of Fig. 3.11 with regular-expression labels

Let us first eliminate state B. Since this state is neither accepting nor the start state, it will not be in any of the reduced automata. Thus, we save work if we eliminate it first, before developing the two reduced automata that correspond to the two accepting states.

State B has one predecessor, A, and one successor, C. In terms of the regular expressions in the diagram of Fig. 3.7: $Q_1 = \mathbf{1}$, $P_1 = \mathbf{0} + \mathbf{1}$, $R_{11} = \emptyset$ (since the arc from A to C does not exist), and $S = \emptyset$ (because there is no loop at state B). As a result, the expression on the new arc from A to C is $\emptyset + \mathbf{1}\emptyset^*(\mathbf{0} + \mathbf{1})$.

To simplify, we first eliminate the initial $\emptyset$, which may be ignored in a union. The expression thus becomes $\mathbf{1}\emptyset^*(\mathbf{0} + \mathbf{1})$. Note that the regular expression $\emptyset^*$ is equivalent to the regular expression ϵ, since

$$L(\emptyset^*) = \{\epsilon\} \cup L(\emptyset) \cup L(\emptyset)L(\emptyset) \cup \cdots$$

Since all the terms but the first are empty, we see that $L(\emptyset^*) = \{\epsilon\}$, which is the same as $L(\epsilon)$. Thus, $\mathbf{1}\emptyset^*(\mathbf{0} + \mathbf{1})$ is equivalent to $\mathbf{1}(\mathbf{0} + \mathbf{1})$, which is the expression we use for the arc $A \to C$ in Fig. 3.13.

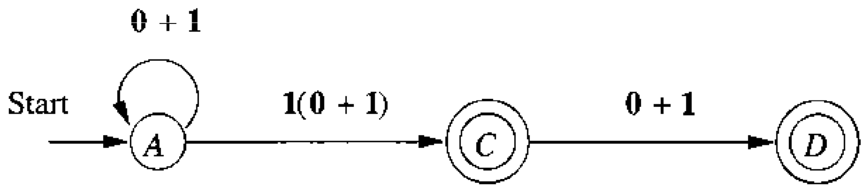


Figure 3.13: Eliminating state B

Now, we must branch, eliminating states C and D in separate reductions. To eliminate state C, the mechanics are similar to those performed above to eliminate state B, and the resulting automaton is shown in Fig. 3.14.

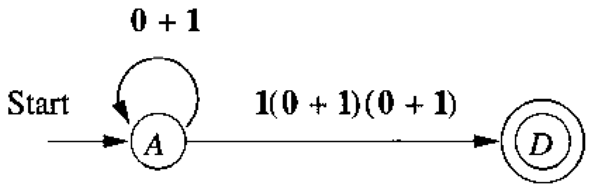


Figure 3.14: A two-state automaton with states A and D

In terms of the generic two-state automaton of Fig. 3.9, the regular expressions from Fig. 3.14 are: $R = \mathbf{0} + \mathbf{1}$, $S = \mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$, $T = \emptyset$, and $U = \emptyset$. The expression U^* can be replaced by ϵ, i.e., eliminated in a concatenation; the justification is that $\emptyset^* = \epsilon$, as we discussed above. Also, the expression SU^*T is equivalent to $\emptyset$, since T, one of the terms of the concatenation, is $\emptyset$. The generic expression $(R + SU^*T)^*SU^*$ thus simplifies in this case to R^*S, or $(\mathbf{0} + \mathbf{1})^*\mathbf{1}(\mathbf{0} + \mathbf{1})(\mathbf{0} + \mathbf{1})$. In informal terms, the language of this expression is any string ending in 1, followed by two symbols that are each either 0 or 1. That language is one portion of the strings accepted by the automaton of Fig. 3.11: those strings whose third position from the end has a 1.

Now, we must start again at Fig. 3.13 and eliminate state D instead of C. Since D has no successors, an inspection of Fig. 3.7 tells us that there will be no changes to arcs, and the arc from C to D is eliminated, along with state D. The resulting two-state automaton is shown in Fig. 3.15.

Ordering the Elimination of States

As we observed in Example 3.6, when a state is neither the start state nor an accepting state, it gets eliminated in all the derived automata. Thus, one of the advantages of the state-elimination process compared with the mechanical generation of regular expressions that we described in Section 3.2.1 is that we can start by eliminating all the states that are neither start nor accepting, once and for all. We only have to begin duplicating the reduction effort when we need to eliminate some accepting states.

Even there, we can combine some of the effort. For instance, if there are three accepting states p, q, and r, we can eliminate p and then branch to eliminate either q or r, thus producing the automata for accepting states r and q, respectively. We then start again with all three accepting states and eliminate both q and r to get the automaton for p.

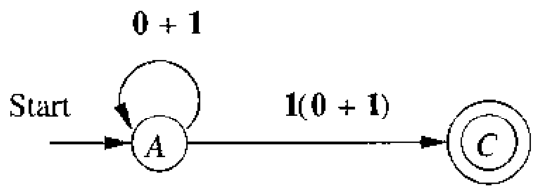


Figure 3.15: Two-state automaton resulting from the elimination of D

This automaton is very much like that of Fig. 3.14; only the label on the arc from the start state to the accepting state is different. Thus, we can apply the rule for two-state automata and simplify the expression to get $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$. This expression represents the other type of string the automaton accepts: those with a 1 in the second position from the end.

All that remains is to sum the two expressions to get the expression for the entire automaton of Fig. 3.11. This expression is

$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1}) + (\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$$

□

3.2.3 Converting Regular Expressions to Automata

We shall now complete the plan of Fig. 3.1 by showing that every language L that is $L(R)$ for some regular expression R, is also $L(E)$ for some ϵ-NFA E. The proof is a structural induction on the expression R. We start by showing how to construct automata for the basis expressions: single symbols, ϵ, and $\emptyset$. We then show how to combine these automata into larger automata that accept the union, concatenation, or closure of the language accepted by smaller automata.

All of the automata we construct are ϵ-NFA's with a single accepting state.

Theorem 3.7: Every language defined by a regular expression is also defined by a finite automaton.

PROOF: Suppose $L = L(R)$ for a regular expression R. We show that $L = L(E)$ for some ϵ-NFA E with:

1. Exactly one accepting state.

2. No arcs into the initial state.

3. No arcs out of the accepting state.

The proof is by structural induction on R, following the recursive definition of regular expressions that we had in Section 3.1.2.

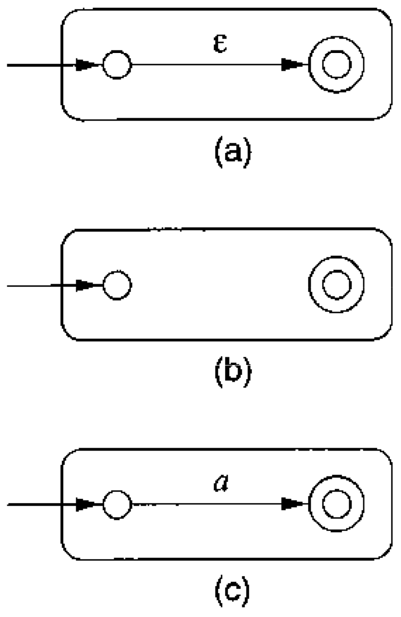


Figure 3.16: The basis of the construction of an automaton from a regular expression

BASIS: There are three parts to the basis, shown in Fig. 3.16. In part (a) we see how to handle the expression ϵ. The language of the automaton is easily seen to be $\{\epsilon\}$, since the only path from the start state to an accepting state is labeled ϵ. Part (b) shows the construction for $\emptyset$. Clearly there are no paths from start state to accepting state, so $\emptyset$ is the language of this automaton. Finally, part (c) gives the automaton for a regular expression **a**. The language of this automaton evidently consists of the one string a, which is also $L(\mathbf{a})$. It is easy to check that these automata all satisfy conditions (1), (2), and (3) of the inductive hypothesis.

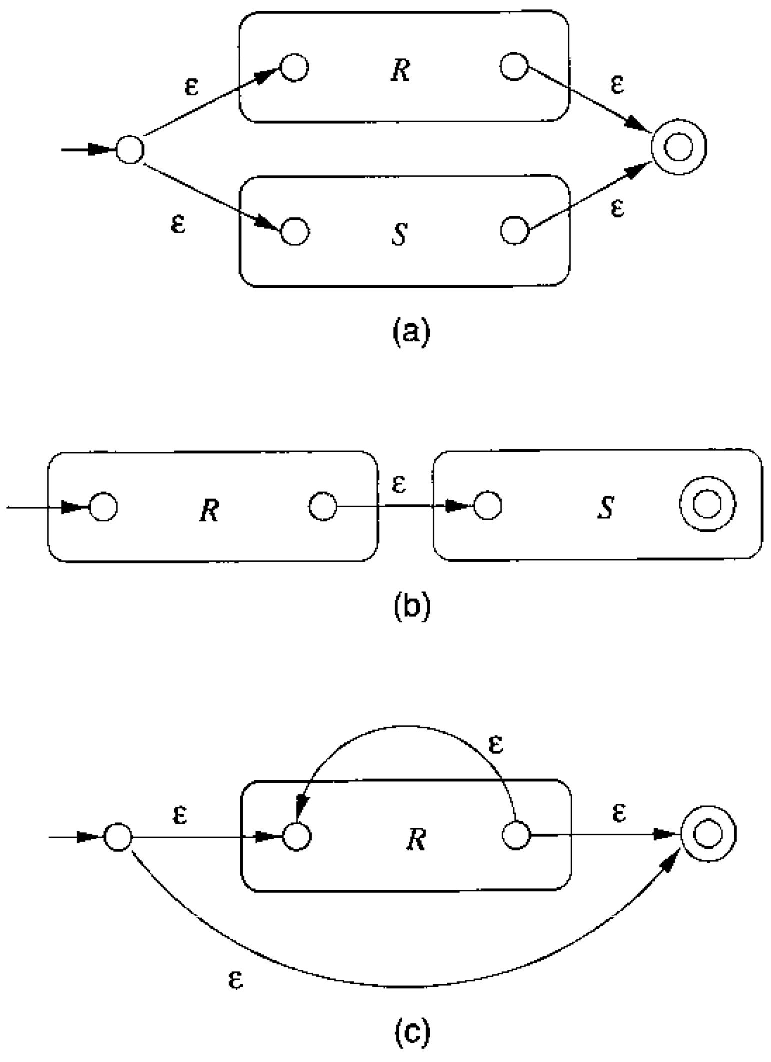


Figure 3.17: The inductive step in the regular-expression-to-ϵ-NFA construction

INDUCTION: The three parts of the induction are shown in Fig. 3.17. We assume that the statement of the theorem is true for the immediate subexpressions of a given regular expression; that is, the languages of these subexpressions are also the languages of ϵ-NFA's with a single accepting state. The four cases are:

1. The expression is $R + S$ for some smaller expressions R and S. Then the automaton of Fig. 3.17(a) serves. That is, starting at the new start state, we can go to the start state of either the automaton for R or the automaton for S. We then reach the accepting state of one of these automata, following a path labeled by some string in $L(R)$ or $L(S)$, respectively. Once we reach the accepting state of the automaton for R or S, we can follow one of the ϵ-arcs to the accepting state of the new automaton.

Thus, the language of the automaton in Fig. 3.17(a) is $L(R) \cup L(S)$.

2. The expression is RS for some smaller expressions R and S. The automaton for the concatenation is shown in Fig. 3.17(b). Note that the start state of the first automaton becomes the start state of the whole, and the accepting state of the second automaton becomes the accepting state of the whole. The idea is that the only paths from start to accepting state go first through the automaton for R, where it must follow a path labeled by a string in $L(R)$, and then through the automaton for S, where it follows a path labeled by a string in $L(S)$. Thus, the paths in the automaton of Fig. 3.17(b) are all and only those labeled by strings in $L(R)L(S)$.

3. The expression is R^* for some smaller expression R. Then we use the automaton of Fig. 3.17(c). That automaton allows us to go either:

   (a) Directly from the start state to the accepting state along a path labeled ϵ. That path lets us accept ϵ, which is in $L(R^*)$ no matter what expression R is.

   (b) To the start state of the automaton for R, through that automaton one or more times, and then to the accepting state. This set of paths allows us to accept strings in $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$, and so on, thus covering all strings in $L(R^*)$ except perhaps ϵ, which was covered by the direct arc to the accepting state mentioned in (3a).

4. The expression is (R) for some smaller expression R. The automaton for R also serves as the automaton for (R), since the parentheses do not change the language defined by the expression.

It is a simple observation that the constructed automata satisfy the three conditions given in the inductive hypothesis — one accepting state, with no arcs into the initial state or out of the accepting state. □

Example 3.8: Let us convert the regular expression $(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})$ to an ϵ-NFA. Our first step is to construct an automaton for $\mathbf{0}+\mathbf{1}$. We use two automata constructed according to Fig. 3.16(c), one with label $\mathbf{0}$ on the arc and one with label $\mathbf{1}$. These two automata are then combined using the union construction of Fig. 3.17(a). The result is shown in Fig. 3.18(a).

Next, we apply to Fig. 3.18(a) the star construction of Fig. 3.17(c). This automaton is shown in Fig. 3.18(b). The last two steps involve applying the concatenation construction of Fig. 3.17(b). First, we connect the automaton of Fig. 3.18(b) to another automaton designed to accept only the string 1. This automaton is another application of the basis construction of Fig. 3.16(c) with label $\mathbf{1}$ on the arc. Note that we must create a *new* automaton to recognize 1; we must not use the automaton for 1 that was part of Fig. 3.18(a). The third automaton in the concatenation is another automaton for $\mathbf{0}+\mathbf{1}$. Again, we must create a copy of the automaton of Fig. 3.18(a); we must not use the same copy that became part of Fig. 3.18(b). The complete automaton is shown in

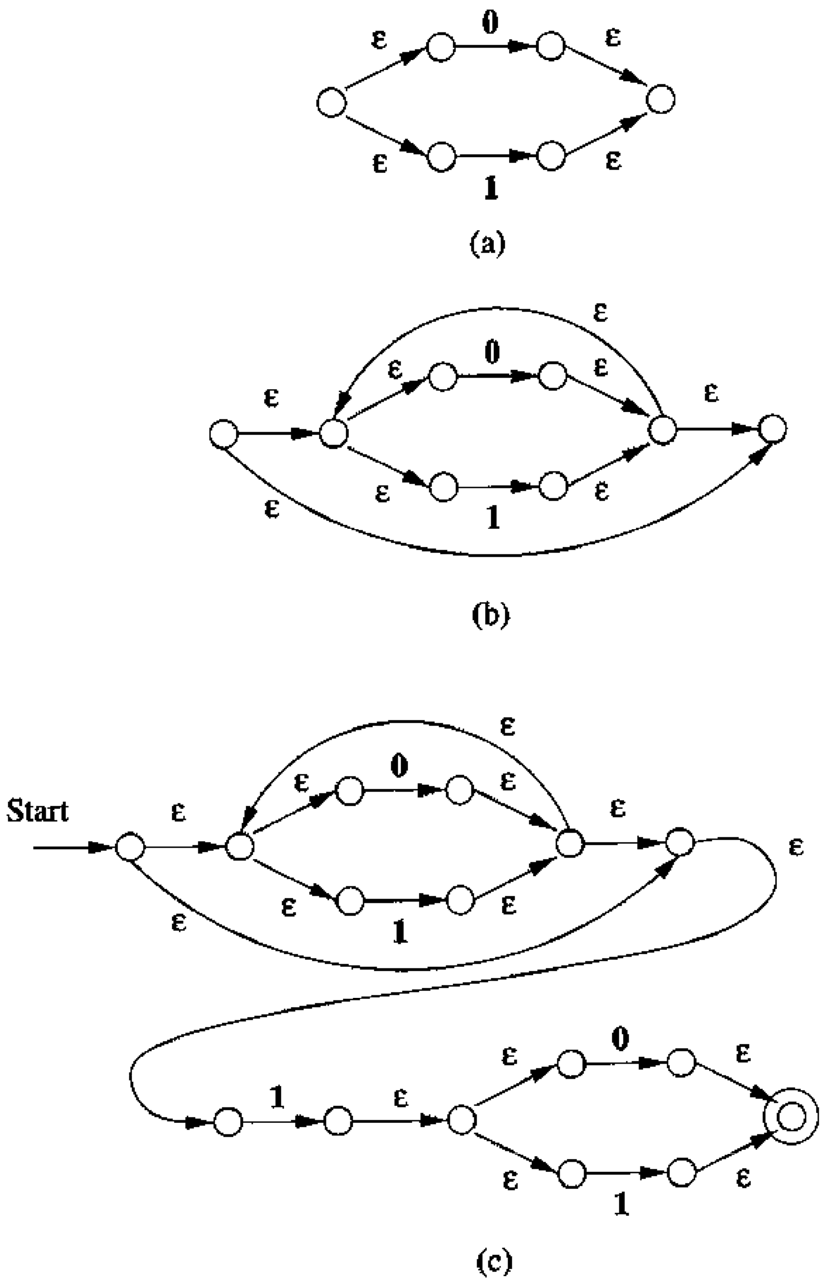


Figure 3.18: Automata constructed for Example 3.8

Fig. 3.18(c). Note that this ϵ-NFA, when ϵ-transitions are removed, looks just like the much simpler automaton of Fig. 3.15 that also accepts the strings that have a 1 in their next-to-last position. □

3.2.4 Exercises for Section 3.2

Exercise 3.2.1: Here is a transition table for a DFA:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

* a) Give all the regular expressions $R_{ij}^{(0)}$. *Note*: Think of state q_i as if it were the state with integer number i.

* b) Give all the regular expressions $R_{ij}^{(1)}$. Try to simplify the expressions as much as possible.

c) Give all the regular expressions $R_{ij}^{(2)}$. Try to simplify the expressions as much as possible.

d) Give a regular expression for the language of the automaton.

* e) Construct the transition diagram for the DFA and give a regular expression for its language by eliminating state q_2.

Exercise 3.2.2: Repeat Exercise 3.2.1 for the following DFA:

	0	1
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

Note that solutions to parts (a), (b) and (e) are *not* available for this exercise.

Exercise 3.2.3: Convert the following DFA to a regular expression, using the state-elimination technique of Section 3.2.2.

	0	1
$\rightarrow *p$	s	p
q	p	s
r	r	q
s	q	r

Exercise 3.2.4: Convert the following regular expressions to NFA's with ϵ-transitions.

* a) $\mathbf{01}^*$.

b) $(\mathbf{0}+\mathbf{1})\mathbf{01}$.

c) $\mathbf{00}(\mathbf{0}+\mathbf{1})^*$.

Exercise 3.2.5: Eliminate ϵ-transitions from your ϵ-NFA's of Exercise 3.2.4. A solution to part (a) appears in the book's Web pages.

! **Exercise 3.2.6:** Let $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ be an ϵ-NFA such that there are no transitions into q_0 and no transitions out of q_f. Describe the language accepted by each of the following modifications of A, in terms of $L = L(A)$:

* a) The automaton constructed from A by adding an ϵ-transition from q_f to q_0.

* b) The automaton constructed from A by adding an ϵ-transition from q_0 to every state reachable from q_0 (along a path whose labels may include symbols of Σ as well as ϵ).

c) The automaton constructed from A by adding an ϵ-transition to q_f from every state that can reach q_f along some path.

d) The automaton constructed from A by doing both (b) and (c).

!! **Exercise 3.2.7:** There are some simplifications to the constructions of Theorem 3.7, where we converted a regular expression to an ϵ-NFA. Here are three:

1. For the union operator, instead of creating new start and accepting states, merge the two start states into one state with all the transitions of both start states. Likewise, merge the two accepting states, having all transitions to either go to the merged state instead.

2. For the concatenation operator, merge the accepting state of the first automaton with the start state of the second.

3. For the closure operator, simply add ϵ-transitions from the accepting state to the start state and vice-versa.

Each of these simplifications, by themselves, still yield a correct construction; that is, the resulting ϵ-NFA for any regular expression accepts the language of the expression. Which subsets of changes (1), (2), and (3) may be made to the construction together, while still yielding a correct automaton for every regular expression?

*!! **Exercise 3.2.8:** Give an algorithm that takes a DFA A and computes the number of strings of length n (for some given n, not related to the number of states of A) accepted by A. Your algorithm should be polynomial in both n and the number of states of A. *Hint*: Use the technique suggested by the construction of Theorem 3.4.

3.3 Applications of Regular Expressions

A regular expression that gives a "picture" of the pattern we want to recognize is the medium of choice for applications that search for patterns in text. The regular expressions are then compiled, behind the scenes, into deterministic or nondeterministic automata, which are then simulated to produce a program that recognizes patterns in text. In this section, we shall consider two important classes of regular-expression-based applications: lexical analyzers and text search.

3.3.1 Regular Expressions in UNIX

Before seeing the applications, we shall introduce the UNIX notation for extended regular expressions. This notation gives us a number of additional capabilities. In fact, the UNIX extensions include certain features, especially the ability to name and refer to previous strings that have matched a pattern, that actually allow nonregular languages to be recognized. We shall not consider these features here; rather we shall only introduce the shorthands that allow complex regular expressions to be written succinctly.

The first enhancement to the regular-expression notation concerns the fact that most real applications deal with the ASCII character set. Our examples have typically used a small alphabet, such as $\{0, 1\}$. The existence of only two symbols allowed us to write succinct expressions like $\mathbf{0} + \mathbf{1}$ for "any character." However, if there were 128 characters, say, the same expression would involve listing them all, and would be highly inconvenient to write. Thus, UNIX regular expressions allow us to write *character classes* to represent large sets of characters as succinctly as possible. The rules for character classes are:

- The symbol `.` (dot) stands for "any character."
- The sequence `[`$a_1a_2 \cdots a_k$`]` stands for the regular expression

  $$a_1 + a_2 + \cdots + a_k$$

  This notation saves about half the characters, since we don't have to write the +-signs. For example, we could express the four characters used in C comparison operators by `[<>=!]`.
- Between the square braces we can put a range of the form x-y to mean all the characters from x to y in the ASCII sequence. Since the digits have codes in order, as do the upper-case letters and the lower-case letters, we can express many of the classes of characters that we really care about with just a few keystrokes. For example, the digits can be expressed `[0-9]`, the upper-case letters can be expressed `[A-Z]`, and the set of all letters and digits can be expressed `[A-Za-z0-9]`. If we want to include a minus sign among a list of characters, we can place it first or last, so it is not confused with its use to form a character range. For example, the set

of digits, plus the dot, plus, and minus signs that are used to form signed decimal numbers may be expressed `[-+.0-9]`. Square brackets, or other characters that have special meanings in UNIX regular expressions can be represented as characters by preceding them with a backslash (\).

- There are special notations for several of the most common classes of characters. For instance:

  a) `[:digit:]` is the set of ten digits, the same as `[0-9]`.[3]

  b) `[:alpha:]` stands for any alphabetic character, as does `[A-Za-z]`.

  c) `[:alnum:]` stands for the digits and letters (alphabetic and numeric characters), as does `[A-Za-z0-9]`.

In addition, there are several operators that are used in UNIX regular expressions that we have not encountered previously. None of these operators extend what languages can be expressed, but they sometimes make it easier to express what we want.

1. The operator | is used in place of $+$ to denote union.

2. The operator ? means "zero or one of." Thus, $R?$ in UNIX is the same as $\epsilon + R$ in this book's regular-expression notation.

3. The operator + means "one or more of." Thus, $R+$ in UNIX is shorthand for RR^* in our notation.

4. The operator $\{n\}$ means "n copies of." Thus, $R\{5\}$ in UNIX is shorthand for $RRRRR$.

Note that UNIX regular expressions allow parentheses to group subexpressions, just as for the regular expressions described in Section 3.1.2, and the same operator precedence is used (with ?, + and $\{n\}$ treated like $*$ as far as precedence is concerned). The star operator $*$ is used in UNIX (without being a superscript, of course) with the same meaning as we have used.

3.3.2 Lexical Analysis

One of the oldest applications of regular expressions was in specifying the component of a compiler called a "lexical analyzer." This component scans the source program and recognizes all *tokens*, those substrings of consecutive characters that belong together logically. Keywords and identifiers are common examples of tokens, but there are many others.

[3]The notation `[:digit:]` has the advantage that should some code other than ASCII be used, including a code where the digits did not have consecutive codes, `[:digit:]` would still represent `[0123456789]`, while `[0-9]` would represent whatever characters had codes between the codes for 0 and 9, inclusive.

The Complete Story for UNIX Regular Expressions

The reader who wants to get the complete list of operators and shorthands available in the UNIX regular-expression notation can find them in the manual pages for various commands. There are some differences among the various versions of UNIX, but a command like `man grep` will get you the notation used for the `grep` command, which is fundamental. "Grep" stands for "Global (search for) Regular Expression and Print," incidentally.

The UNIX command `lex` and its GNU version `flex`, accept as input a list of regular expressions, in the UNIX style, each followed by a bracketed section of code that indicates what the lexical analyzer is to do when it finds an instance of that token. Such a facility is called a ***lexical-analyzer generator***, because it takes as input a high-level description of a lexical analyzer and produces from it a function that is a working lexical analyzer.

Commands such as `lex` and `flex` have been found extremely useful because the regular-expression notation is exactly as powerful as we need to describe tokens. These commands are able to use the regular-expression-to-DFA conversion process to generate an efficient function that breaks source programs into tokens. They make the implementation of a lexical analyzer an afternoon's work, while before the development of these regular-expression-based tools, the hand-generation of the lexical analyzer could take months. Further, if we need to modify the lexical analyzer for any reason, it is often a simple matter to change a regular expression or two, instead of having to go into mysterious code to fix a bug.

Example 3.9: In Fig. 3.19 is an example of partial input to the `lex` command, describing some of the tokens that are found in the language C. The first line handles the keyword `else` and the action is to return a symbolic constant (`ELSE` in this example) to the parser for further processing. The second line contains a regular expression describing identifiers: a letter followed by zero or more letters and/or digits. The action is first to enter that identifier in the symbol table if not already there; `lex` isolates the token found in a buffer, so this piece of code knows exactly what identifier was found. Finally, the lexical analyzer returns the symbolic constant `ID`, which has been chosen in this example to represent identifiers.

The third entry in Fig. 3.19 is for the sign `>=`, a two-character operator. The last example we show is for the sign `=`, a one-character operator. There would in practice appear expressions describing each of the keywords, each of the signs and punctuation symbols like commas and parentheses, and families of constants such as numbers and strings. Many of these are very simple, just a sequence of one or more specific characters. However, some have more

```
else                   {return(ELSE);}

[A-Za-z][A-Za-z0-9]*   {code to enter the found identifier
                        in the symbol table;
                        return(ID);
                       }

>=                     {return(GE);}

=                      {return(EQ);}

...
```

Figure 3.19: A sample of `lex` input

of the flavor of identifiers, requiring the full power of the regular-expression notation to describe. The integers, floating-point numbers, character strings, and comments are other examples of sets of strings that profit from the regular-expression capabilities of commands like `lex`. □

The conversion of a collection of expressions, such as those suggested in Fig. 3.19, to an automaton proceeds approximately as we have described formally in the preceding sections. We start by building an automaton for the union of all the expressions. This automaton in principle tells us only that *some* token has been recognized. However, if we follow the construction of Theorem 3.7 for the union of expressions, the ϵ-NFA state tells us exactly which token has been recognized.

The only problem is that more than one token may be recognized at once; for instance, the string `else` matches not only the regular expression **else** but also the expression for identifiers. The standard resolution is for the lexical-analyzer generator to give priority to the first expression listed. Thus, if we want keywords like `else` to be *reserved* (not usable as identifiers), we simply list them ahead of the expression for identifiers.

3.3.3 Finding Patterns in Text

In Section 2.4.1 we introduced the notion that automata could be used to search efficiently for a set of words in a large repository such as the Web. While the tools and technology for doing so are not so well developed as that for lexical analyzers, the regular-expression notation is valuable for describing searches for interesting patterns. As for lexical analyzers, the capability to go from the natural, descriptive regular-expression notation to an efficient (automaton-based) implementation offers substantial intellectual leverage.

The general problem for which regular-expression technology has been found useful is the description of a vaguely defined class of patterns in text. The vagueness of the description virtually guarantees that we shall not describe the pattern correctly at first — perhaps we can never get exactly the right description. By using regular-expression notation, it becomes easy to describe the patterns at a high level, with little effort, and to modify the description quickly when things go wrong. A "compiler" for regular expressions is useful to turn the expressions we write into executable code.

Let us explore an extended example of the sort of problem that arises in many Web applications. Suppose that we want to scan a very large number of Web pages and detect addresses. We might simply want to create a mailing list. Or, perhaps we are trying to classify businesses by their location so that we can answer queries like "find me a restaurant within 10 minutes drive of where I am now."

We shall focus on recognizing street addresses in particular. What is a street address? We'll have to figure that out, and if, while testing the software, we find we miss some cases, we'll have to modify the expressions to capture what we were missing. To begin, a street address will probably end in "Street" or its abbreviation, "St." However, some people live on "Avenues" or "Roads," and these might be abbreviated in the address as well. Thus, we might use as the ending for our regular expression something like:

```
Street|St\.|Avenue|Ave\.|Road|Rd\.
```

In the above expression, we have used UNIX-style notation, with the vertical bar, rather than +, as the union operator. Note also that the dots are *escaped* with a preceding backslash, since dot has the special meaning of "any character" in UNIX expressions, and in this case we really want only the period or "dot" character to end the three abbreviations.

The designation such as `Street` must be preceded by the name of the street. Usually, the name is a capital letter followed by some lower-case letters. We can describe this pattern by the UNIX expression `[A-Z][a-z]*`. However, some streets have a name consisting of more than one word, such as Rhode Island Avenue in Washington DC. Thus, after discovering that we were missing addresses of this form, we could revise our description of street names to be

```
'[A-Z][a-z]*( [A-Z][a-z]*)*'
```

The expression above starts with a group consisting of a capital and zero or more lower-case letters. There follow zero or more groups consisting of a blank, another capital letter, and zero or more lower-case letters. The blank is an ordinary character in UNIX expressions, but to avoid having the above expression look like two expressions separated by a blank in a UNIX command line, we are required to place quotation marks around the whole expression. The quotes are not part of the expression itself.

Now, we need to include the house number as part of the address. Most house numbers are a string of digits. However, some will have a letter following, as in "123A Main St." Thus, the expression we use for numbers has an optional capital letter following: `[0-9]+[A-Z]?`. Notice that we use the UNIX + operator for "one or more" digits and the ? operator for "zero or one" capital letter. The entire expression we have developed for street addresses is:

```
'[0-9]+[A-Z]? [A-Z][a-z]*( [A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd\.)'
```

If we work with this expression, we shall do fairly well. However, we shall eventually discover that we are missing:

1. Streets that are called something other than a street, avenue, or road. For example, we shall miss "Boulevard," "Place," "Way," and their abbreviations.

2. Street names that are numbers, or partially numbers, like "42nd Street."

3. Post-Office boxes and rural-delivery routes.

4. Street names that don't end in anything like "Street." An example is El Camino Real in Silicon Valley. Being Spanish for "the royal road," saying "El Camino Real Road" would be redundant, so one has to deal with complete addresses like "2000 El Camino Real."

5. All sorts of strange things we can't even imagine. Can you?

Thus, having a regular-expression compiler can make the process of slow convergence to the complete recognizer for addresses much easier than if we had to recode every change directly in a conventional programming language.

3.3.4 Exercises for Section 3.3

! **Exercise 3.3.1:** Give a regular expression to describe phone numbers in all the various forms you can think of. Consider international numbers as well as the fact that different countries have different numbers of digits in area codes and in local phone numbers.

!! **Exercise 3.3.2:** Give a regular expression to represent salaries as they might appear in employment advertising. Consider that salaries might be given on a per hour, week, month, or year basis. They may or may not appear with a dollar sign, or other unit such as "K" following. There may be a word or words nearby that identify a salary. Suggestion: look at classified ads in a newspaper, or on-line jobs listings to get an idea of what patterns might be useful.

! **Exercise 3.3.3:** At the end of Section 3.3.3 we gave some examples of improvements that could be possible for the regular expression that describes addresses. Modify the expression developed there to include all the mentioned options.

3.4 Algebraic Laws for Regular Expressions

In Example 3.5, we saw the need for simplifying regular expressions, in order to keep the size of expressions manageable. There, we gave some ad-hoc arguments why one expression could be replaced by another. In all cases, the basic issue was that the two expressions were *equivalent*, in the sense that they defined the same languages. In this section, we shall offer a collection of algebraic laws that bring to a higher level the issue of when two regular expressions are equivalent. Instead of examining specific regular expressions, we shall consider pairs of regular expressions with variables as arguments. Two expressions with variables are *equivalent* if whatever languages we substitute for the variables, the results of the two expressions are the same language.

An example of this process in the algebra of arithmetic is as follows. It is one matter to say that $1+2 = 2+1$. That is an example of the commutative law of addition, and it is easy to check by applying the addition operator on both sides and getting $3 = 3$. However, the *commutative law of addition* says more; it says that $x + y = y + x$, where x and y are variables that can be replaced by any two numbers. That is, no matter what two numbers we add, we get the same result regardless of the order in which we sum them.

Like arithmetic expressions, the regular expressions have a number of laws that work for them. Many of these are similar to the laws for arithmetic, if we think of union as addition and concatenation as multiplication. However, there are a few places where the analogy breaks down, and there are also some laws that apply to regular expressions but have no analog for arithmetic, especially when the closure operator is involved. The next sections form a catalog of the major laws. We conclude with a discussion of how one can check whether a proposed law for regular expressions is indeed a law; i.e., it will hold for any languages that we may substitute for the variables.

3.4.1 Associativity and Commutativity

Commutativity is the property of an operator that says we can switch the order of its operands and get the same result. An example for arithmetic was given above: $x + y = y + x$. *Associativity* is the property of an operator that allows us to regroup the operands when the operator is applied twice. For example, the associative law of multiplication is $(x \times y) \times z = x \times (y \times z)$. Here are three laws of these types that hold for regular expressions:

- $L + M = M + L$. This law, the *commutative law for union*, says that we may take the union of two languages in either order.

- $(L + M) + N = L + (M + N)$. This law, the *associative law for union*, says that we may take the union of three languages either by taking the union of the first two initially, or taking the union of the last two initially. Note that, together with the commutative law for union, we conclude that we can take the union of any collection of languages with any order and grouping, and the result will be the same. Intuitively, a string is in $L_1 \cup L_2 \cup \cdots \cup L_k$ if and only if it is in one or more of the L_i's.

- $(LM)N = L(MN)$. This law, the ***associative law for concatenation***, says that we can concatenate three languages by concatenating either the first two or the last two initially.

Missing from this list is the "law" $LM = ML$, which would say that concatenation is commutative. However, this law is false.

Example 3.10: Consider the regular expressions **01** and **10**. These expressions denote the languages $\{01\}$ and $\{10\}$, respectively. Since the languages are different the general law $LM = ML$ cannot hold. If it did, we could substitute the regular expression **0** for L and **1** for M and conclude falsely that $\mathbf{01} = \mathbf{10}$. □

3.4.2 Identities and Annihilators

An *identity* for an operator is a value such that when the operator is applied to the identity and some other value, the result is the other value. For instance, 0 is the identity for addition, since $0 + x = x + 0 = x$, and 1 is the identity for multiplication, since $1 \times x = x \times 1 = x$. An *annihilator* for an operator is a value such that when the operator is applied to the annihilator and some other value, the result is the annihilator. For instance, 0 is an annihilator for multiplication, since $0 \times x = x \times 0 = 0$. There is no annihilator for addition.

There are three laws for regular expressions involving these concepts; we list them below.

- $\emptyset + L = L + \emptyset = L$. This law asserts that $\emptyset$ is the identity for union.

- $\epsilon L = L\epsilon = L$. This law asserts that ϵ is the identity for concatenation.

- $\emptyset L = L\emptyset = \emptyset$. This law asserts that $\emptyset$ is the annihilator for concatenation.

These laws are powerful tools in simplifications. For example, if we have a union of several expressions, some of which are, or have been simplified to $\emptyset$, then the $\emptyset$'s can be dropped from the union. Likewise, if we have a concatenation of several expressions, some of which are, or have been simplified to ϵ, we can drop the ϵ's from the concatenation. Finally, if we have a concatenation of any number of expressions, and even one of them is $\emptyset$, then the entire concatenation can be replaced by $\emptyset$.

3.4.3 Distributive Laws

A *distributive law* involves two operators, and asserts that one operator can be pushed down to be applied to each argument of the other operator individually. The most common example from arithmetic is the distributive law of multiplication over addition, that is, $x \times (y + z) = x \times y + x \times z$. Since multiplication is commutative, it doesn't matter whether the multiplication is on the left or right of the sum. However, there is an analogous law for regular expressions, that we must state in two forms, since concatenation is not commutative. These laws are:

- $L(M+N) = LM + LN$. This law, is the ***left distributive law of concatenation over union.***
- $(M+N)L = ML + NL$. This law, is the ***right distributive law of concatenation over union.***

Let us prove the left distributive law; the other is proved similarly. The proof will refer to languages only; it does not depend on the languages having regular expressions.

Theorem 3.11: If L, M, and N are any languages, then

$$L(M \cup N) = LM \cup LN$$

PROOF: The proof is similar to another proof about a distributive law that we saw in Theorem 1.10. We need first to show that a string w is in $L(M \cup N)$ if and only if it is in $LM \cup LN$.

(Only if) If w is in $L(M \cup N)$, then $w = xy$, where x is in L and y is in either M or N. If y is in M, then xy is in LM, and therefore in $LM \cup LN$. Likewise, if y is in N, then xy is in LN and therefore in $LM \cup LN$.

(If) Suppose w is in $LM \cup LN$. Then w is in either LM or in LN. Suppose first that w is in LM. Then $w = xy$, where x is in L and y is in M. As y is in M, it is also in $M \cup N$. Thus, xy is in $L(M \cup N)$. If w is not in LM, then it is surely in LN, and a similar argument shows it is in $L(M \cup N)$. □

Example 3.12: Consider the regular expression $\mathbf{0}+\mathbf{01}^*$. We can "factor out a $\mathbf{0}$" from the union, but first we have to recognize that the expression $\mathbf{0}$ by itself is actually the concatenation of $\mathbf{0}$ with something, namely ϵ. That is, we use the identity law for concatenation to replace $\mathbf{0}$ by $\mathbf{0}\epsilon$, giving us the expression $\mathbf{0}\epsilon + \mathbf{01}^*$. Now, we can apply the left distributive law to replace this expression by $\mathbf{0}(\epsilon + \mathbf{1}^*)$. If we further recognize that ϵ is in $L(\mathbf{1}^*)$, then we observe that $\epsilon + \mathbf{1}^* = \mathbf{1}^*$, and can simplify to $\mathbf{01}^*$. □

3.4.4 The Idempotent Law

An operator is said to be *idempotent* if the result of applying it to two of the same values as arguments is that value. The common arithmetic operators are not idempotent; $x + x \neq x$ in general and $x \times x \neq x$ in general (although there are *some* values of x for which the equality holds, such as $0 + 0 = 0$). However, union and intersection are common examples of idempotent operators. Thus, for regular expressions, we may assert the following law:

- $L + L = L$. This law, the ***idempotence law for union***, states that if we take the union of two identical expressions, we can replace them by one copy of the expression.

3.4.5 Laws Involving Closures

There are a number of laws involving the closure operators and its UNIX-style variants $^+$ and ?. We shall list them here, and give some explanation for why they are true.

- $(L^*)^* = L^*$. This law says that closing an expression that is already closed does not change the language. The language of $(L^*)^*$ is all strings created by concatenating strings in the language of L^*. But those strings are themselves composed of strings from L. Thus, the string in $(L^*)^*$ is also a concatenation of strings from L and is therefore in the language of L^*.

- $\emptyset^* = \epsilon$. The closure of $\emptyset$ contains only the string ϵ, as we discussed in Example 3.6.

- $\epsilon^* = \epsilon$. It is easy to check that the only string that can be formed by concatenating any number of copies of the empty string is the empty string itself.

- $L^+ = LL^* = L^*L$. Recall that L^+ is defined to be $L + LL + LLL + \cdots$. Also, $L^* = \epsilon + L + LL + LLL + \cdots$. Thus,

  $$LL^* = L\epsilon + LL + LLL + LLLL + \cdots$$

  When we remember that $L\epsilon = L$, we see that the infinite expansions for LL^* and for L^+ are the same. That proves $L^+ = LL^*$. The proof that $L^+ = L^*L$ is similar.[4]

- $L^* = L^+ + \epsilon$. The proof is easy, since the expansion of L^+ includes every term in the expansion of L^* except ϵ. Note that if the language L contains the string ϵ, then the additional "$+\epsilon$" term is not needed; that is, $L^+ = L^*$ in this special case.

- $L? = \epsilon + L$. This rule is really the definition of the ? operator.

3.4.6 Discovering Laws for Regular Expressions

Each of the laws above was proved, formally or informally. However, there is an infinite variety of laws about regular expressions that might be proposed. Is there a general methodology that will make our proofs of the correct laws

[4]Notice that, as a consequence, any language L commutes (under concatenation) with its own closure; $LL^* = L^*L$. That rule does not contradict the fact that, in general, concatenation is not commutative.

easy? It turns out that the truth of a law reduces to a question of the equality of two specific languages. Interestingly, the technique is closely tied to the regular-expression operators, and cannot be extended to expressions involving some other operators, such as intersection.

To see how this test works, let us consider a proposed law, such as

$$(L+M)^* = (L^*M^*)^*$$

This law says that if we have any two languages L and M, and we close their union, we get the same language as if we take the language L^*M^*, that is, all strings composed of zero or more choices from L followed by zero or more choices from M, and close that language.

To prove this law, suppose first that string w is in the language of $(L+M)^*$.[5] Then we can write $w = w_1w_2\cdots w_k$ for some k, where each w_i is in either L or M. It follows that each w_i is in the language of L^*M^*. To see why, if w is in L, pick one string, w_i, from L; this string is also in L^*. Pick no strings from M; that is, pick ϵ from M^*. If w_i is in M, the argument is similar. Once every w_i is seen to be in L^*M^*, it follows that w is in the closure of this language.

To complete the proof, we also have to prove the converse: that strings in $(L^*M^*)^*$ are also in $(L+M)^*$. We omit this part of the proof, since our objective is not to prove the law, but to notice the following important property of regular expressions.

Any regular expression with variables can be thought of as a *concrete* regular expression, one that has no variables, by thinking of each variable as if it were a distinct symbol. For example, the expression $(L+M)^*$ can have variables L and M replaced by symbols a and b, respectively, giving us the regular expression $(\mathbf{a}+\mathbf{b})^*$.

The language of the concrete expression guides us regarding the form of strings in any language that is formed from the original expression when we replace the variables by languages. Thus, in our analysis of $(L+M)^*$, we observed that any string w composed of a sequence of choices from either L or M, would be in the language of $(L+M)^*$. We can arrive at that conclusion by looking at the language of the concrete expression, $L\big((\mathbf{a}+\mathbf{b})^*\big)$, which is evidently the set of all strings of a's and b's. We could substitute any string in L for any occurrence of a in one of those strings, and we could substitute any string in M for any occurrence of b, with possibly different choices of strings for different occurrences of a or b. Those substitutions, applied to all the strings in $(\mathbf{a}+\mathbf{b})^*$, gives us all strings formed by concatenating strings from L and/or M, in any order.

The above statement may seem obvious, but as is pointed out in the box on "Extensions of the Test Beyond Regular Expressions May Fail," it is not even true when some other operators are added to the three regular-expression operators. We prove the general principle for regular expressions in the next theorem.

[5]For simplicity, we shall identify the regular expressions and their languages, and avoid saying "the language of" in front of every regular expression.

Theorem 3.13: Let E be a regular expression with variables $L_1, L_2, \ldots, L_m$. Form concrete regular expression C by replacing each occurrence of L_i by the symbol a_i, for $i = 1, 2, \ldots, m$. Then for any languages $L_1, L_2, \ldots, L_m$, every string w in $L(E)$ can be written $w = w_1 w_2 \cdots w_k$, where each w_i is in one of the languages, say L_{j_i}, and the string $a_{j_1} a_{j_2} \cdots a_{j_k}$ is in the language $L(C)$. Less formally, we can construct $L(E)$ by starting with each string in $L(C)$, say $a_{j_1} a_{j_2} \cdots a_{j_k}$, and substituting for each of the a_{j_i}'s any string from the corresponding language L_{j_i}.

PROOF: The proof is a structural induction on the expression E.

BASIS: The basis cases are where E is ϵ, $\emptyset$, or a variable L. In the first two cases, there is nothing to prove, since the concrete expression C is the same as E. If E is a variable L, then $L(E) = L$. The concrete expression C is just **a**, where a is the symbol corresponding to L. Thus, $L(C) = \{a\}$. If we substitute any string in L for the symbol a in this one string, we get the language L, which is also $L(E)$.

INDUCTION: There are three cases, depending on the final operator of E. First, suppose that $E = F + G$; i.e., a union is the final operator. Let C and D be the concrete expressions formed from F and G, respectively, by substituting concrete symbols for the language-variables in these expressions. Note that the same symbol must be substituted for all occurrences of the same variable, in both F and G. Then the concrete expression that we get from E is $C + D$, and $L(C + D) = L(C) + L(D)$.

Suppose that w is a string in $L(E)$, when the language variables of E are replaced by specific languages. Then w is in either $L(F)$ or $L(G)$. By the inductive hypothesis, w is obtained by starting with a concrete string in $L(C)$ or $L(D)$, respectively, and substituting for the symbols strings in the corresponding languages. Thus, in either case, the string w can be constructed by starting with a concrete string in $L(C+D)$, and making the same substitutions of strings for symbols.

We must also consider the cases where E is FG or F^*. However, the arguments are similar to the union case above, and we leave them for you to complete. $\square$

3.4.7 The Test for a Regular-Expression Algebraic Law

Now, we can state and prove the test for whether or not a law of regular expressions is true. The test for whether $E = F$ is true, where E and F are two regular expressions with the same set of variables, is:

1. Convert E and F to concrete regular expressions C and D, respectively, by replacing each variable by a concrete symbol.

2. Test whether $L(C) = L(D)$. If so, then $E = F$ is a true law, and if not, then the "law" is false. Note that we shall not see the test for whether two

regular expressions denote the same language until Section 4.4. However, we can use ad-hoc means to decide the equality of the pairs of languages that we actually care about. Recall that if the languages are ***not*** the same, than it is sufficient to provide one counterexample: a single string that is in one language but not the other.

Theorem 3.14: The above test correctly identifies the true laws for regular expressions.

PROOF: We shall show that $L(E) = L(F)$ for any languages in place of the variables of E and F if and only if $L(C) = L(D)$.

(Only-if) Suppose $L(E) = L(F)$ for all choices of languages for the variables. In particular, choose for every variable L the concrete symbol a that replaces L in expressions C and D. Then for this choice, $L(C) = L(E)$, and $L(D) = L(F)$. Since $L(E) = L(F)$ is given, it follows that $L(C) = L(D)$.

(If) Suppose $L(C) = L(D)$. By Theorem 3.13, $L(E)$ and $L(F)$ are each constructed by replacing the concrete symbols of strings in $L(C)$ and $L(D)$, respectively, by strings in the languages that correspond to those symbols. If the strings of $L(C)$ and $L(D)$ are the same, then the two languages constructed in this manner will also be the same; that is, $L(E) = L(F)$. $\square$

Example 3.15: Consider the prospective law $(L + M)^* = (L^*M^*)^*$. If we replace variables L and M by concrete symbols a and b respectively, we get the regular expressions $(\mathbf{a} + \mathbf{b})^*$ and $(\mathbf{a}^*\mathbf{b}^*)^*$. It is easy to check that both these expressions denote the language with all strings of a's and b's. Thus, the two concrete expressions denote the same language, and the law holds.

For another example of a law, consider $L^* = L^*L^*$. The concrete languages are $\mathbf{a}^*$ and $\mathbf{a}^*\mathbf{a}^*$, respectively, and each of these is the set of all strings of a's. Again, the law is found to hold; that is, concatenation of a closed language with itself yields that language.

Finally, consider the prospective law $L + ML = (L + M)L$. If we choose symbols a and b for variables L and M, respectively, we have the two concrete regular expressions $\mathbf{a} + \mathbf{ba}$ and $(\mathbf{a} + \mathbf{b})\mathbf{a}$. However, the languages of these expressions are not the same. For example, the string aa is in the second, but not the first. Thus, the prospective law is false. $\square$

3.4.8 Exercises for Section 3.4

Exercise 3.4.1: Verify the following identities involving regular expressions.

* a) $R + S = S + R$.

b) $(R + S) + T = R + (S + T)$.

c) $(RS)T = R(ST)$.

Extensions of the Test Beyond Regular Expressions May Fail

Let us consider an extended regular-expression algebra that includes the intersection operator. Interestingly, adding $\cap$ to the three regular-expression operators does not increase the set of languages we can describe, as we shall see in Theorem 4.8. However, it does make the test for algebraic laws invalid.

Consider the "law" $L \cap M \cap N = L \cap M$; that is, the intersection of any three languages is the same as the intersection of the first two of these languages. This "law" is patently false. For example, let $L = M = \{a\}$ and $N = \emptyset$. But the test based on concretizing the variables would fail to see the difference. That is, if we replaced L, M, and N by the symbols a, b, and c, respectively, we would test whether $\{a\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. Since both sides are the empty set, the equality of languages holds and the test would imply that the "law" is true.

d) $R(S+T) = RS + RT$.

e) $(R+S)T = RT + ST$.

* f) $(R^*)^* = R^*$.

g) $(\epsilon + R)^* = R^*$.

h) $(R^*S^*)^* = (R+S)^*$.

! **Exercise 3.4.2:** Prove or disprove each of the following statements about regular expressions.

* a) $(R+S)^* = R^* + S^*$.

b) $(RS+R)^*R = R(SR+R)^*$.

* c) $(RS+R)^*RS = (RR^*S)^*$.

d) $(R+S)^*S = (R^*S)^*$.

e) $S(RS+S)^*R = RR^*S(RR^*S)^*$.

Exercise 3.4.3: In Example 3.6, we developed the regular expression

$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1}) + (\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1})(\mathbf{0}+\mathbf{1})$$

Use the distributive laws to develop two different, simpler, equivalent expressions.

Exercise 3.4.4: At the beginning of Section 3.4.6, we gave part of a proof that $(L^*M^*)^* = (L+M)^*$. Complete the proof by showing that strings in $(L^*M^*)^*$ are also in $(L+M)^*$.

! **Exercise 3.4.5:** Complete the proof of Theorem 3.13 by handling the cases where regular expression E is of the form FG or of the form F^*.

3.5 Summary of Chapter 3

- ✦ *Regular Expressions*: This algebraic notation describes exactly the same languages as finite automata: the regular languages. The regular-expression operators are union, concatenation (or "dot"), and closure (or "star").

- ✦ *Regular Expressions in Practice*: Systems such as UNIX and various of its commands use an extended regular-expression language that provides shorthands for many common expressions. Character classes allow the easy expression of sets of symbols, while operators such as one-or-more-of and at-most-one-of augment the usual regular-expression operators.

- ✦ *Equivalence of Regular Expressions and Finite Automata*: We can convert a DFA to a regular expression by an inductive construction in which expressions for the labels of paths allowed to pass through increasingly larger sets of states are constructed. Alternatively, we can use a state-elimination procedure to build the regular expression for a DFA. In the other direction, we can construct recursively an ϵ-NFA from regular expressions, and then convert the ϵ-NFA to a DFA, if we wish.

- ✦ *The Algebra of Regular Expressions*: Regular expressions obey many of the algebraic laws of arithmetic, although there are differences. Union and concatenation are associative, but only union is commutative. Concatenation distributes over union. Union is idempotent.

- ✦ *Testing Algebraic Identities*: We can tell whether a regular-expression equivalence involving variables as arguments is true by replacing the variables by distinct constants and testing whether the resulting languages are the same.

3.6 References for Chapter 3

The idea of regular expressions and the proof of their equivalence to finite automata is the work of S. C. Kleene [3]. However, the construction of an ϵ-NFA from a regular expression, as presented here, is the "McNaughton-Yamada construction," from [4]. The test for regular-expression identities by treating variables as constants was written down by J. Gischer [2]. Although thought to be folklore, this report demonstrated how adding several other operations such as intersection or shuffle (See Exercise 7.3.4) makes the test fail, even though they do not extend the class of languages representable.

Even before developing UNIX, K. Thompson was investigating the use of regular expressions in commands such as `grep`, and his algorithm for processing such commands appears in [5]. The early development of UNIX produced several other commands that make heavy use of the extended regular-expression notation, such as M. Lesk's `lex` command. A description of this command and other regular-expression techniques can be found in [1].

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1986.

2. J. L. Gischer, STAN-CS-TR-84-1033 (1984).

3. S. C. Kleene, "Representation of events in nerve nets and finite automata," In C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956, pp. 3–42.

4. R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IEEE Trans. Electronic Computers* **9**:1 (Jan., 1960), pp. 39–47.

5. K. Thompson, "Regular expression search algorithm," *Comm. ACM* **11**:6 (June, 1968), pp. 419–422.

Chapter 4

Properties of Regular Languages

The chapter explores the properties of regular languages. Our first tool for this exploration is a way to prove that certain languages are not regular. This theorem, called the "pumping lemma," is introduced in Section 4.1.

One important kind of fact about the regular languages is called a "closure property." These properties let us build recognizers for languages that are constructed from other languages by certain operations. As an example, the intersection of two regular languages is also regular. Thus, given automata that recognize two different regular languages, we can construct mechanically an automaton that recognizes exactly the intersection of these two languages. Since the automaton for the intersection may have many more states than either of the two given automata, this "closure property" can be a useful tool for building complex automata. Section 2.1 used this construction in an essential way.

Some other important facts about regular languages are called "decision properties." Our study of these properties gives us algorithms for answering important questions about automata. A central example is an algorithm for deciding whether two automata define the same language. A consequence of our ability to decide this question is that we can "minimize" automata, that is, find an equivalent to a given automaton that has as few states as possible. This problem has been important in the design of switching circuits for decades, since the cost of the circuit (area of a chip that the circuit occupies) tends to decrease as the number of states of the automaton implemented by the circuit decreases.

4.1 Proving Languages not to be Regular

We have established that the class of languages known as the regular languages has at least four different descriptions. They are the languages accepted by DFA's, by NFA's, and by ϵ-NFA's; they are also the languages defined by regular expressions.

Not every language is a regular language. In this section, we shall introduce a powerful technique, known as the "pumping lemma," for showing certain languages not to be regular. We then give several examples of nonregular languages. In Section 4.2 we shall see how the pumping lemma can be used in tandem with closure properties of the regular languages to prove other languages not to be regular.

4.1.1 The Pumping Lemma for Regular Languages

Let us consider the language $L_{01} = \{0^n1^n \mid n \geq 1\}$. This language contains all strings $01, 0011, 000111$, and so on, that consist of one or more 0's followed by an equal number of 1's. We claim that L_{01} is not a regular language. The intuitive argument is that if L_{01} were regular, then L_{01} would be the language of some DFA A. This automaton has some particular number of states, say k states. Imagine this automaton receiving k 0's as input. It is in some state after consuming each of the $k+1$ prefixes of the input: $\epsilon, 0, 00, \ldots, 0^k$. Since there are only k different states, the pigeonhole principle tells us that after reading two different prefixes, say 0^i and 0^j, A must be in the same state, say state q.

However, suppose instead that after reading i or j 0's, the automaton A starts receiving 1's as input. After receiving i 1's, it must accept if it previously received i 0's, but not if it received j 0's. Since it was in state q when the 1's started, it cannot "remember" whether it received i or j 0's, so we can "fool" A and make it do the wrong thing — accept if it should not, or fail to accept when it should.

The above argument is informal, but can be made precise. However, the same conclusion, that the language L_{01} is not regular, can be reached using a general result, as follows.

Theorem 4.1: (The *pumping lemma for regular languages*) Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that:

1. $y \neq \epsilon$.

2. $|xy| \leq n$.

3. For all $k \geq 0$, the string xy^kz is also in L.

That is, we can always find a nonempty string y not too far from the beginning of w that can be "pumped"; that is, repeating y any number of times, or deleting it (the case $k = 0$), keeps the resulting string in the language L.

PROOF: Suppose L is regular. Then $L = L(A)$ for some DFA A. Suppose A has n states. Now, consider any string w of length n or more, say $w = a_1a_2 \cdots a_m$, where $m \geq n$ and each a_i is an input symbol. For $i = 0, 1, \ldots, n$ define state p_i to be $\hat{\delta}(q_0, a_1a_2 \cdots a_i)$, where δ is the transition function of A, and q_0 is the start state of A. That is, p_i is the state A is in after reading the first i symbols of w. Note that $p_0 = q_0$.

By the pigeonhole principle, it is not possible for the $n + 1$ different p_i's for $i = 0, 1, \ldots, n$ to be distinct, since there are only n different states. Thus, we can find two different integers i and j, with $0 \leq i < j \leq n$, such that $p_i = p_j$. Now, we can break $w = xyz$ as follows:

1. $x = a_1a_2 \cdots a_i$.

2. $y = a_{i+1}a_{i+2} \cdots a_j$.

3. $z = a_{j+1}a_{j+2} \cdots a_m$.

That is, x takes us to p_i once; y takes us from p_i back to p_i (since p_i is also p_j), and z is the balance of w. The relationships among the strings and states are suggested by Fig. 4.1. Note that x may be empty, in the case that $i = 0$. Also, z may be empty if $j = n = m$. However, y can not be empty, since i is strictly less than j.

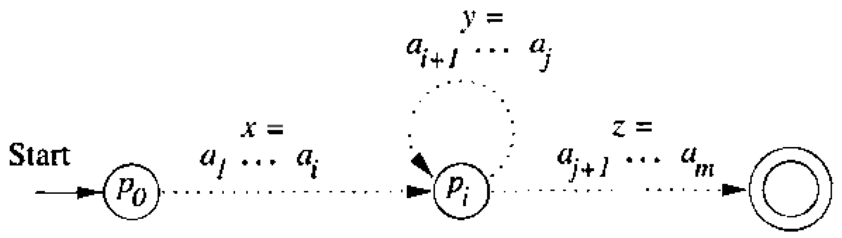


Figure 4.1: Every string longer than the number of states must cause a state to repeat

Now, consider what happens if the automaton A receives the input xy^kz for any $k \geq 0$. If $k = 0$, then the automaton goes from the start state q_0 (which is also p_0) to p_i on input x. Since p_i is also p_j, it must be that A goes from p_i to the accepting state shown in Fig. 4.1 on input z. Thus, A accepts xz.

If $k > 0$, then A goes from q_0 to p_i on input x, circles from p_i to p_i k times on input y^k, and then goes to the accepting state on input z. Thus, for any $k \geq 0$, xy^kz is also accepted by A; that is, xy^kz is in L. $\square$

4.1.2 Applications of the Pumping Lemma

Let us see some examples of how the pumping lemma is used. In each case, we shall propose a language and use the pumping lemma to prove that the language is not regular.

The Pumping Lemma as an Adversarial Game

Recall our discussion from Section 1.2.3 where we pointed out that a theorem whose statement involves several alternations of "for-all" and "there-exists" quantifiers can be thought of as a game between two players. The pumping lemma is an important example of this type of theorem, since it in effect involves four different quantifiers: "**for all** regular languages L **there exists** n such that **for all** w in L with $|w| \geq n$ **there exists** xyz equal to w such that $\cdots$." We can see the application of the pumping lemma as a game, in which:

1. Player 1 picks the language L to be proved nonregular.
2. Player 2 picks n, but doesn't reveal to player 1 what n is; player 1 must devise a play for all possible n's.
3. Player 1 picks w, which may depend on n and which must be of length at least n.
4. Player 2 divides w into x, y, and z, obeying the constraints that are stipulated in the pumping lemma; $y \neq \epsilon$ and $|xy| \leq n$. Again, player 2 does not have to tell player 1 what x, y, and z are, although they must obey the constraints.
5. Player 1 "wins" by picking k, which may be a function of n, x, y, and z, such that xy^kz is not in L.

Example 4.2 : Let us show that the language L_{eq} consisting of all strings with an equal number of 0's and 1's (not in any particular order) is not a regular language. In terms of the "two-player game" described in the box on "The Pumping Lemma as an Adversarial Game," we shall be player 1 and we must deal with whatever choices player 2 makes. Suppose n is the constant that must exist if L_{eq} is regular, according to the pumping lemma; i.e., "player 2" picks n. We shall pick $w = 0^n1^n$, that is, n 0's followed by n 1's, a string that surely is in L_{eq}.

Now, "player 2" breaks our w up into xyz. All we know is that $y \neq \epsilon$, and $|xy| \leq n$. However, that information is very useful, and we "win" as follows. Since $|xy| \leq n$, and xy comes at the front of w, we know that x and y consist only of 0's. The pumping lemma tells us that xz is in L_{eq}, if L_{eq} is regular. This conclusion is the case $k = 0$ in the pumping lemma.[1] However, xz has n 1's, since all the 1's of w are in z. But xz also has fewer than n 0's, because we

[1]Observe in what follows that we could have also succeeded by picking $k = 2$, or indeed any value of k other than 1.

lost the 0's of y. Since $y \neq \epsilon$ we know that there can be no more than $n-1$ 0's among x and z. Thus, after assuming L_{eq} is a regular language, we have proved a fact known to be false, that xz is in L_{eq}. We have a proof by contradiction of the fact that L_{eq} is not regular. □

Example 4.3: Let us show that the language L_{pr} consisting of all strings of 1's whose length is a prime is not a regular language. Suppose it were. Then there would be a constant n satisfying the conditions of the pumping lemma. Consider some prime $p \geq n+2$; there must be such a p, since there are an infinity of primes. Let $w = 1^p$.

By the pumping lemma, we can break $w = xyz$ such that $y \neq \epsilon$ and $|xy| \leq n$. Let $|y| = m$. Then $|xz| = p - m$. Now consider the string $xy^{p-m}z$, which must be in L_{pr} by the pumping lemma, if L_{pr} really is regular. However,

$$|xy^{p-m}z| = |xz| + (p-m)|y| = p - m + (p-m)m = (m+1)(p-m)$$

It looks like $|xy^{p-m}z|$ is not a prime, since it has two factors $m+1$ and $p-m$. However, we must check that neither of these factors are 1, since then $(m+1)(p-m)$ might be a prime after all. But $m+1 > 1$, since $y \neq \epsilon$ tells us $m \geq 1$. Also, $p - m > 1$, since $p \geq n+2$ was chosen, and $m \leq n$ since

$$m = |y| \leq |xy| \leq n$$

Thus, $p - m \geq 2$.

Again we have started by assuming the language in question was regular, and we derived a contradiction by showing that some string not in the language was required by the pumping lemma to be in the language. Thus, we conclude that L_{pr} is not a regular language. □

4.1.3 Exercises for Section 4.1

Exercise 4.1.1: Prove that the following are not regular languages.

a) $\{0^n1^n \mid n \geq 1\}$. This language, consisting of a string of 0's followed by an equal-length string of 1's, is the language L_{01} we considered informally at the beginning of the section. Here, you should apply the pumping lemma in the proof.

b) The set of strings of balanced parentheses. These are the strings of characters "(" and ")" that can appear in a well-formed arithmetic expression.

* c) $\{0^n10^n \mid n \geq 1\}$.

d) $\{0^n1^m2^n \mid n$ and m are arbitrary integers$\}$.

e) $\{0^n1^m \mid n \leq m\}$.

f) $\{0^n1^{2n} \mid n \geq 1\}$.

! **Exercise 4.1.2:** Prove that the following are not regular languages.

* a) $\{0^n \mid n \text{ is a perfect square}\}$.

b) $\{0^n \mid n \text{ is a perfect cube}\}$.

c) $\{0^n \mid n \text{ is a power of } 2\}$.

d) The set of strings of 0's and 1's whose length is a perfect square.

e) The set of strings of 0's and 1's that are of the form ww, that is, some string repeated.

f) The set of strings of 0's and 1's that are of the form ww^R, that is, some string followed by its reverse. (See Section 4.2.2 for a formal definition of the reversal of a string.)

g) The set of strings of 0's and 1's of the form $w\overline{w}$, where $\overline{w}$ is formed from w by replacing all 0's by 1's, and vice-versa; e.g., $\overline{011} = 100$, and 011100 is an example of a string in the language.

h) The set of strings of the form $w1^n$, where w is a string of 0's and 1's of length n.

! **Exercise 4.1.3:** Prove that the following are not regular languages.

a) The set of strings of 0's and 1's, beginning with a 1, such that when interpreted as an integer, that integer is a prime.

b) The set of strings of the form 0^i1^j such that the greatest common divisor of i and j is 1.

! **Exercise 4.1.4:** When we try to apply the pumping lemma to a regular language, the "adversary wins," and we cannot complete the proof. Show what goes wrong when we choose L to be one of the following languages:

* a) The empty set.

* b) $\{00, 11\}$.

* c) $(\mathbf{00} + \mathbf{11})^*$.

d) $\mathbf{01^*0^*1}$.

4.2 Closure Properties of Regular Languages

In this section, we shall prove several theorems of the form "if certain languages are regular, and a language L is formed from them by certain operations (e.g., L is the union of two regular languages), then L is also regular." These theorems are often called *closure properties* of the regular languages, since they show that the class of regular languages is closed under the operation mentioned. Closure properties express the idea that when one (or several) languages are regular, then certain related languages are also regular. They also serve as an interesting illustration of how the equivalent representations of the regular languages (automata and regular expressions) reinforce each other in our understanding of the class of languages, since often one representation is far better than the others in supporting a proof of a closure property. Here is a summary of the principal closure properties for regular languages:

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure (star) of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular.

4.2.1 Closure of Regular Languages Under Boolean Operations

Our first closure properties are the three boolean operations: union, intersection, and complementation:

1. Let L and M be languages over alphabet Σ. Then $L \cup M$ is the language that contains all strings that are in either or both of L and M.
2. Let L and M be languages over alphabet Σ. Then $L \cap M$ is the language that contains all strings that are in both L and M.
3. Let L be a language over alphabet Σ. Then $\overline{L}$, the *complement* of L, is the set of strings in Σ^* that are not in L.

It turns out that the regular languages are closed under all three of the boolean operations. The proofs take rather different approaches though, as we shall see.

What if Languages Have Different Alphabets?

When we take the union or intersection of two languages L and M, they might have different alphabets. For example, it is possible that $L_1 \subseteq \{a, b\}$ while $L_2 \subseteq \{b, c, d\}$. However, if a language L consists of strings with symbols in Σ, then we can also think of L as a language over any finite alphabet that is a superset of Σ. Thus, for example, we can think of both L_1 and L_2 above as being languages over alphabet $\{a, b, c, d\}$. The fact that none of L_1's strings contain symbols c or d is irrelevant, as is the fact that L_2's strings will not contain a.

Likewise, when taking the complement of a language L that is a subset of Σ_1^* for some alphabet Σ_1, we may choose to take the complement *with respect to* some alphabet Σ_2 that is a superset of Σ_1. If so, then the complement of L will be $\Sigma_2^* - L$; that is, the complement of L with respect to Σ_2 includes (among other strings) all those strings in Σ_2^* that have at least one symbol that is in Σ_2 but not in Σ_1. Had we taken the complement of L with respect to Σ_1, then no string with symbols in $\Sigma_2 - \Sigma_1$ would be in $\overline{L}$. Thus, to be strict, we should always state the alphabet with respect to which a complement is taken. However, often it is obvious which alphabet is meant; e.g., if L is defined by an automaton, then the specification of that automaton includes the alphabet. Thus, we shall often speak of the "complement" without specifying the alphabet.

Closure Under Union

Theorem 4.4: If L and M are regular languages, then so is $L \cup M$.

PROOF: This proof is simple. Since L and M are regular, they have regular expressions; say $L = L(R)$ and $M = L(S)$. Then $L \cup M = L(R + S)$ by the definition of the + operator for regular expressions. $\Box$

Closure Under Complementation

The theorem for union was made very easy by the use of the regular-expression representation for the languages. However, let us next consider complementation. Do you see how to take a regular expression and change it into one that defines the complement language? Well neither do we. However, it can be done, because as we shall see in Theorem 4.5, it is easy to start with a DFA and construct a DFA that accepts the complement. Thus, starting with a regular expression, we could find a regular expression for its complement as follows:

1. Convert the regular expression to an ϵ-NFA.

2. Convert that ϵ-NFA to a DFA by the subset construction.

Closure Under Regular Operations

The proof that regular languages are closed under union was exceptionally easy because union is one of the three operations that define the regular expressions. The same idea as Theorem 4.4 applies to concatenation and closure as well. That is:

- If L and M are regular languages, then so is LM.
- If L is a regular language, then so is L^*.

3. Complement the accepting states of that DFA.
4. Turn the complement DFA back into a regular expression using the construction of Sections 3.2.1 or 3.2.2.

Theorem 4.5: If L is a regular language over alphabet Σ, then $\overline{L} = \Sigma^* - L$ is also a regular language.

PROOF: Let $L = L(A)$ for some DFA $A = (Q, \Sigma, \delta, q_0, F)$. Then $\overline{L} = L(B)$, where B is the DFA $(Q, \Sigma, \delta, q_0, Q - F)$. That is, B is exactly like A, but the accepting states of A have become nonaccepting states of B, and vice versa. Then w is in $L(B)$ if and only if $\hat{\delta}(q_0, w)$ is in $Q - F$, which occurs if and only if w is not in $L(A)$. □

Notice that it is important for the above proof that $\hat{\delta}(q_0, w)$ is always some state; i.e., there are no missing transitions in A. If there were, then certain strings might lead neither to an accepting nor nonaccepting state of A, and those strings would be missing from both $L(A)$ and $L(B)$. Fortunately, we have defined a DFA to have a transition on every symbol of Σ from every state, so each string leads either to a state in F or a state in $Q - F$.

Example 4.6: Let A be the automaton of Fig. 2.14. Recall that DFA A accepts all and only the strings of 0's and 1's that end in 01; in regular-expression terms, $L(A) = (\mathbf{0} + \mathbf{1})^*\mathbf{01}$. The complement of $L(A)$ is therefore all strings of 0's and 1's that do *not* end in 01. Figure 4.2 shows the automaton for $\{0, 1\}^* - L(A)$. It is the same as Fig. 2.14 but with the accepting state made nonaccepting and the two nonaccepting states made accepting. □

Example 4.7: In this example, we shall apply Theorem 4.5 to show a certain language not to be regular. In Example 4.2 we showed that the language L_{eq} consisting of strings with an equal number of 0's and 1's and is not regular. This proof was a straightforward application of the pumping lemma. Now consider

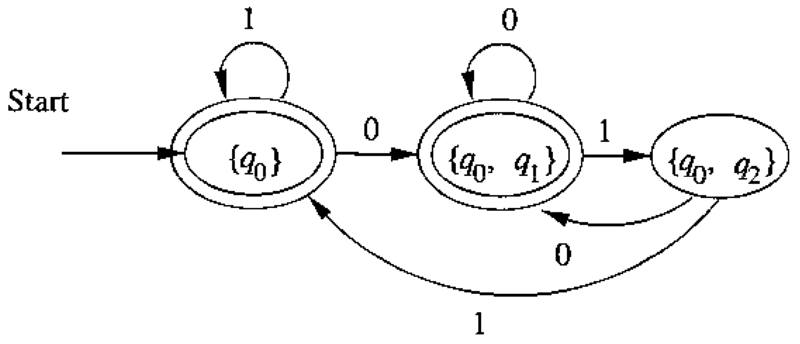


Figure 4.2: DFA accepting the complement of the language $(\mathbf{0}+\mathbf{1})^*\mathbf{01}$

the language M consisting of those strings of 0's and 1's that have an unequal number of 0's and 1's.

It would be hard to use the pumping lemma to show M is not regular. Intuitively, if we start with some string w in M, break it into $w = xyz$, and "pump" y, we might find that y itself was a string like 01 that had an equal number of 0's and 1's. If so, then for no k will xy^kz have an equal number of 0's and 1's, since xyz has an unequal number of 0's and 1's, and the numbers of 0's and 1's change equally as we "pump" y. Thus, we can never use the pumping lemma to contradict the assumption that M is regular.

However, M is still not regular. The reason is that $M = \overline{L}$. Since the complement of the complement is the set we started with, it also follows that $L = \overline{M}$. If M is regular, then by Theorem 4.5, L is regular. But we know L is *not* regular, so we have a proof by contradiction that M is not regular. □

Closure Under Intersection

Now, let us consider the intersection of two regular languages. We actually have little to do, since the three boolean operations are not independent. Once we have ways of performing complementation and union, we can obtain the intersection of languages L and M by the identity

$$L \cap M = \overline{\overline{L} \cup \overline{M}} \tag{4.1}$$

In general, the intersection of two sets is the set of elements that are not in the complement of either set. That observation, which is what Equation (4.1) says, is one of *DeMorgan's laws.* The other law is the same with union and intersection interchanged; that is, $L \cup M = \overline{\overline{L} \cap \overline{M}}$.

However, we can also perform a direct construction of a DFA for the intersection of two regular languages. This construction, which essentially runs two DFA's in parallel, is useful in its own right. For instance, we used it to construct the automaton in Fig. 2.3 that represented the "product" of what two participants — the bank and the store — were doing. We shall make the *product construction* formal in the next theorem.

Theorem 4.8: If L and M are regular languages, then so is $L \cap M$.

PROOF: Let L and M be the languages of automata $A_L = (Q_L, \Sigma, \delta_L, q_L, F_L)$ and $A_M = (Q_M, \Sigma, \delta_M, q_M, F_M)$. Notice that we are assuming that the alphabets of both automata are the same; that is, Σ is the union of the alphabets of L and M, if those alphabets are different. The product construction actually works for NFA's as well as DFA's, but to make the argument as simple as possible, we assume that A_L and A_M are DFA's.

For $L \cap M$ we shall construct an automaton A that simulates both A_L and A_M. The states of A are pairs of states, the first from A_L and the second from A_M. To design the transitions of A, suppose A is in state (p, q), where p is the state of A_L and q is the state of A_M. If a is the input symbol, we see what A_L does on input a; say it goes to state s. We also see what A_M does on input a; say it makes a transition to state t. Then the next state of A will be (s, t). In that manner, A has simulated the effect of both A_L and A_M. The idea is sketched in Fig. 4.3.

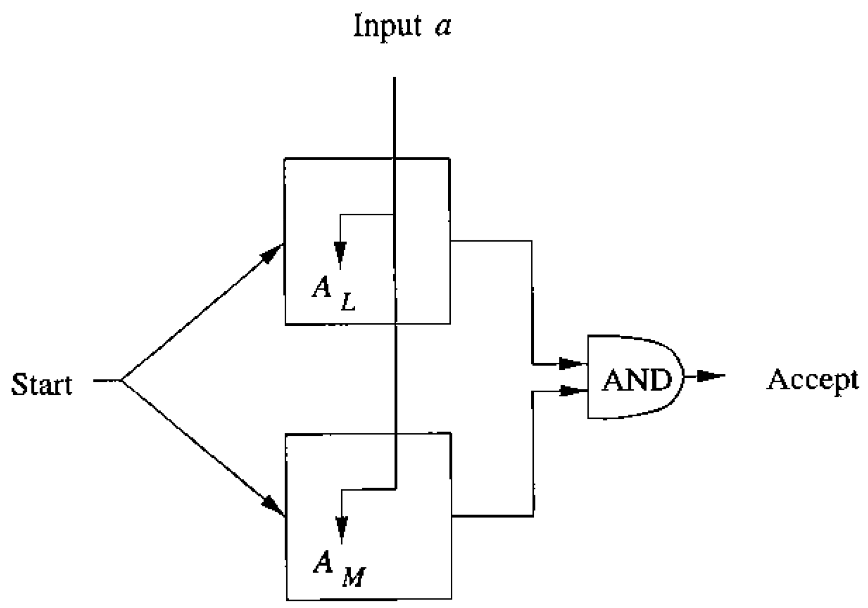


Figure 4.3: An automaton simulating two other automata and accepting if and only if both accept

The remaining details are simple. The start state of A is the pair of start states of A_L and A_M. Since we want to accept if and only if both automata accept, we select as the accepting states of A all those pairs (p, q) such that p is an accepting state of A_L and q is an accepting state of A_M. Formally, we define:

$$A = (Q_L \times Q_M, \Sigma, \delta, (q_L, q_M), F_L \times F_M)$$

where $\delta\big((p, q), a\big) = \big(\delta_L(p, a), \delta_M(q, a)\big)$.

To see why $L(A) = L(A_L) \cap L(A_M)$, first observe that an easy induction on $|w|$ proves that $\hat{\delta}\big((q_L, q_M), w\big) = \big(\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w)\big)$. But A accepts w if and only if $\hat{\delta}\big((q_L, q_M), w\big)$ is a pair of accepting states. That is, $\hat{\delta}_L(q_L, w)$ must be in F_L, and $\hat{\delta}_M(q_M, w)$ must be in F_M. Put another way, w is accepted by A if and only if both A_L and A_M accept w. Thus, A accepts the intersection of L and M. □

Example 4.9: In Fig. 4.4 we see two DFA's. The automaton in Fig. 4.4(a) accepts all those strings that have a 0, while the automaton in Fig. 4.4(b) accepts all those strings that have a 1. We show in Fig. 4.4(c) the product of these two automata. Its states are labeled by the pairs of states of the automata in (a) and (b).

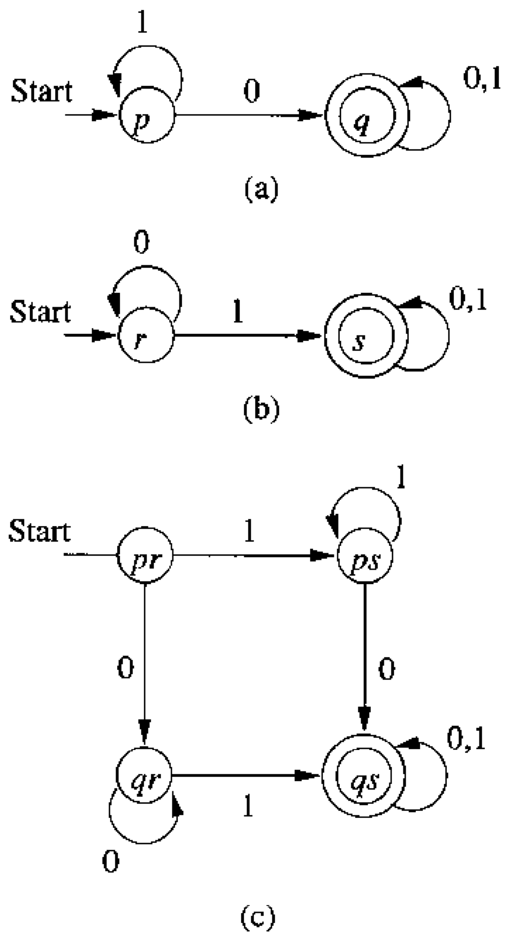


Figure 4.4: The product construction

It is easy to argue that this automaton accepts the intersection of the first two languages: those strings that have both a 0 and a 1. State pr represents only the initial condition, in which we have seen neither 0 nor 1. State qr means that we have seen only 0's, while state ps represents the condition that we have seen only 1's. The accepting state qs represents the condition where we have seen both 0's and 1's. □

Closure Under Difference

There is a fourth operation that is often applied to sets and is related to the boolean operations: set difference. In terms of languages, $L - M$, the *difference* of L and M, is the set of strings that are in language L but not in language M. The regular languages are also closed under this operation, and the proof follows easily from the theorems just proven.

Theorem 4.10: If L and M are regular languages, then so is $L - M$.

PROOF: Observe that $L - M = L \cap \overline{M}$. By Theorem 4.5, $\overline{M}$ is regular, and by Theorem 4.8 $L \cap \overline{M}$ is regular. Therefore $L - M$ is regular. □

4.2.2 Reversal

The *reversal* of a string $a_1 a_2 \cdots a_n$ is the string written backwards, that is, $a_n a_{n-1} \cdots a_1$. We use w^R for the reversal of string w. Thus, 0010^R is 0100, and $\epsilon^R = \epsilon$.

The reversal of a language L, written L^R, is the language consisting of the reversals of all its strings. For instance, if $L = \{001, 10, 111\}$, then $L^R = \{100, 01, 111\}$.

Reversal is another operation that preserves regular languages; that is, if L is a regular language, so is L^R. There are two simple proofs, one based on automata and one based on regular expressions. We shall give the automaton-based proof informally, and let you fill in the details if you like. We then prove the theorem formally using regular expressions.

Given a language L that is $L(A)$ for some finite automaton, perhaps with nondeterminism and ϵ-transitions, we may construct an automaton for L^R by:

1. Reverse all the arcs in the transition diagram for A.

2. Make the start state of A be the only accepting state for the new automaton.

3. Create a new start state p_0 with transitions on ϵ to all the accepting states of A.

The result is an automaton that simulates A "in reverse," and therefore accepts a string w if and only if A accepts w^R. Now, we prove the reversal theorem formally.

Theorem 4.11: If L is a regular language, so is L^R.

PROOF: Assume L is defined by regular expression E. The proof is a structural induction on the size of E. We show that there is another regular expression E^R such that $L(E^R) = \big(L(E)\big)^R$; that is, the language of E^R is the reversal of the language of E.

BASIS: If E is ϵ, $\emptyset$, or **a**, for some symbol a, then E^R is the same as E. That is, we know $\{\epsilon\}^R = \{\epsilon\}$, $\emptyset^R = \emptyset$, and $\{a\}^R = \{a\}$.

INDUCTION: There are three cases, depending on the form of E.

1. $E = E_1 + E_2$. Then $E^R = E_1^R + E_2^R$. The justification is that the reversal of the union of two languages is obtained by computing the reversals of the two languages and taking the union of those languages.

2. $E = E_1E_2$. Then $E^R = E_2^RE_1^R$. Note that we reverse the order of the two languages, as well as reversing the languages themselves. For instance, if $L(E_1) = \{01, 111\}$ and $L(E_2) = \{00, 10\}$, then $L(E_1E_2) = \{0100, 0110, 11100, 11110\}$. The reversal of the latter language is

$$\{0010, 0110, 00111, 01111\}$$

If we concatenate the reversals of $L(E_2)$ and $L(E_1)$ in that order, we get

$$\{00, 01\}\{10, 111\} = \{0010, 00111, 0110, 01111\}$$

which is the same language as $\big(L(E_1E_2)\big)^R$. In general, if a word w in $L(E)$ is the concatenation of w_1 from $L(E_1)$ and w_2 from $L(E_2)$, then $w^R = w_2^Rw_1^R$.

3. $E = E_1^*$. Then $E^R = (E_1^R)^*$. The justification is that any string w in $L(E)$ can be written as $w_1w_2\cdots w_n$, where each w_i is in $L(E)$. But

$$w^R = w_n^Rw_{n-1}^R\cdots w_1^R$$

Each w_i^R is in $L(E^R)$, so w^R is in $L\big((E_1^R)^*\big)$. Conversely, any string in $L\big((E_1^R)^*\big)$ is of the form $w_1w_2\cdots w_n$, where each w_i is the reversal of a string in $L(E_1)$. The reversal of this string, $w_n^Rw_{n-1}^R\cdots w_1^R$, is therefore a string in $L(E_1^*)$, which is $L(E)$. We have thus shown that a string is in $L(E)$ if and only if its reversal is in $L\big((E_1^R)^*\big)$.

□

Example 4.12: Let L be defined by the regular expression $(\mathbf{0} + \mathbf{1})\mathbf{0}^*$. Then L^R is the language of $(\mathbf{0}^*)^R(\mathbf{0} + \mathbf{1})^R$, by the rule for concatenation. If we apply the rules for closure and union to the two parts, and then apply the basis rule that says the reversals of **0** and **1** are unchanged, we find that L^R has regular expression $\mathbf{0}^*(\mathbf{0} + \mathbf{1})$. □

4.2.3 Homomorphisms

A string *homomorphism* is a function on strings that works by substituting a particular string for each symbol.

Example 4.13: The function h defined by $h(0) = ab$ and $h(1) = \epsilon$ is a homomorphism. Given any string of 0's and 1's, it replaces all 0's by the string ab and replaces all 1's by the empty string. For example, h applied to the string 0011 is $abab$. □

Formally, if h is a homomorphism on alphabet Σ, and $w = a_1a_2\cdots a_n$ is a string of symbols in Σ, then $h(w) = h(a_1)h(a_2)\cdots h(a_n)$. That is, we apply h to each symbol of w and concatenate the results, in order. For instance, if h is the homomorphism in Example 4.13, and $w = 0011$, then $h(w) = h(0)h(0)h(1)h(1) = (ab)(ab)(\epsilon)(\epsilon) = abab$, as we claimed in that example.

Further, we can apply a homomorphism to a language by applying it to each of the strings in the language. That is, if L is a language over alphabet Σ, and h is a homomorphism on Σ, then $h(L) = \{h(w) \mid w \text{ is in } L\}$. For instance, if L is the language of regular expression $\mathbf{10^*1}$, i.e., any number of 0's surrounded by single 1's, then $h(L)$ is the language $(\mathbf{ab})^*$. The reason is that h of Example 4.13 effectively drops the 1's, since they are replaced by ϵ, and turns each 0 into ab. The same idea, applying the homomorphism directly to the regular expression, can be used to prove that the regular languages are closed under homomorphisms.

Theorem 4.14: If L is a regular language over alphabet Σ, and h is a homomorphism on Σ, then $h(L)$ is also regular.

PROOF: Let $L = L(R)$ for some regular expression R. In general, if E is a regular expression with symbols in Σ, let $h(E)$ be the expression we obtain by replacing each symbol a of Σ in E by $h(a)$. We claim that $h(R)$ defines the language $h(L)$.

The proof is an easy structural induction that says whenever we take a subexpression E of R and apply h to it to get $h(E)$, the language of $h(E)$ is the same language we get if we apply h to the language $L(E)$. Formally, $L\big(h(E)\big) = h\big(L(E)\big)$.

BASIS: If E is ϵ or $\emptyset$, then $h(E)$ is the same as E, since h does not affect the string ϵ or the language $\emptyset$. Thus, $L\big(h(E)\big) = L(E)$. However, if E is $\emptyset$ or ϵ, then $L(E)$ contains either no strings or a string with no symbols, respectively. Thus $h\big(L(E)\big) = L(E)$ in either case. We conclude $L\big(h(E)\big) = L(E) = h\big(L(E)\big)$.

The only other basis case is if $E = \mathbf{a}$ for some symbol a in Σ. In this case, $L(E) = \{a\}$, so $h\big(L(E)\big) = \{h(a)\}$. Also, $h(E)$ is the regular expression that is the string of symbols $h(a)$. Thus, $L\big(h(E)\big)$ is also $\{h(a)\}$, and we conclude $L\big(h(E)\big) = h\big(L(E)\big)$.

INDUCTION: There are three cases, each of them simple. We shall prove only the union case, where $E = F+G$. The way we apply homomorphisms to regular expressions assures us that $h(E) = h(F+G) = h(F) + h(G)$. We also know that $L(E) = L(F) \cup L(G)$ and

$$L\big(h(E)\big) = L\big(h(F) + h(G)\big) = L\big(h(F)\big) \cup L\big(h(G)\big) \tag{4.2}$$

by the definition of what "+" means in regular expressions. Finally,

$$h\big(L(E)\big) = h\big(L(F) \cup L(G)\big) = h\big(L(F)\big) \cup h\big(L(G)\big) \tag{4.3}$$

because h is applied to a language by application to each of its strings individually. Now we may invoke the inductive hypothesis to assert that $L\big(h(F)\big) = h\big(L(F)\big)$ and $L\big(h(G)\big) = h\big(L(G)\big)$. Thus, the final expressions in (4.2) and (4.3) are equivalent, and therefore so are their respective first terms; that is, $L\big(h(E)\big) = h\big(L(E)\big)$.

We shall not prove the cases where expression E is a concatenation or closure; the ideas are similar to the above in both cases. The conclusion is that $L\big(h(R)\big)$ is indeed $h\big(L(R)\big)$; i.e., applying the homomorphism h to the regular expression for language L results in a regular expression that defines the language $h(L)$. $\Box$

4.2.4 Inverse Homomorphisms

Homomorphisms may also be applied "backwards," and in this mode they also preserve regular languages. That is, suppose h is a homomorphism from some alphabet Σ to strings in another (possibly the same) alphabet T.[2] Let L be a language over alphabet T. Then $h^{-1}(L)$, read "h inverse of L," is the set of strings w in Σ^* such that $h(w)$ is in L. Figure 4.5 suggests the effect of a homomorphism on a language L in part (a), and the effect of an inverse homomorphism in part (b).

Example 4.15: Let L be the language of regular expression $(\mathbf{00} + \mathbf{1})^*$. That is, L consists of all strings of 0's and 1's such that all the 0's occur in adjacent pairs. Thus, 0010011 and 10000111 are in L, but 000 and 10100 are not.

Let h be the homomorphism defined by $h(a) = 01$ and $h(b) = 10$. We claim that $h^{-1}(L)$ is the language of regular expression $(\mathbf{ba})^*$, that is, all strings of repeating ba pairs. We shall prove that $h(w)$ is in L if and only if w is of the form $baba \cdots ba$.

(If) Suppose w is n repetitions of ba for some $n \geq 0$. Note that $h(ba) = 1001$, so $h(w)$ is n repetitions of 1001. Since 1001 is composed of two 1's and a pair of 0's, we know that 1001 is in L. Therefore any repetition of 1001 is also formed from 1 and 00 segments and is in L. Thus, $h(w)$ is in L.

[2]That "T" should be thought of as a Greek capital tau, the letter following sigma.

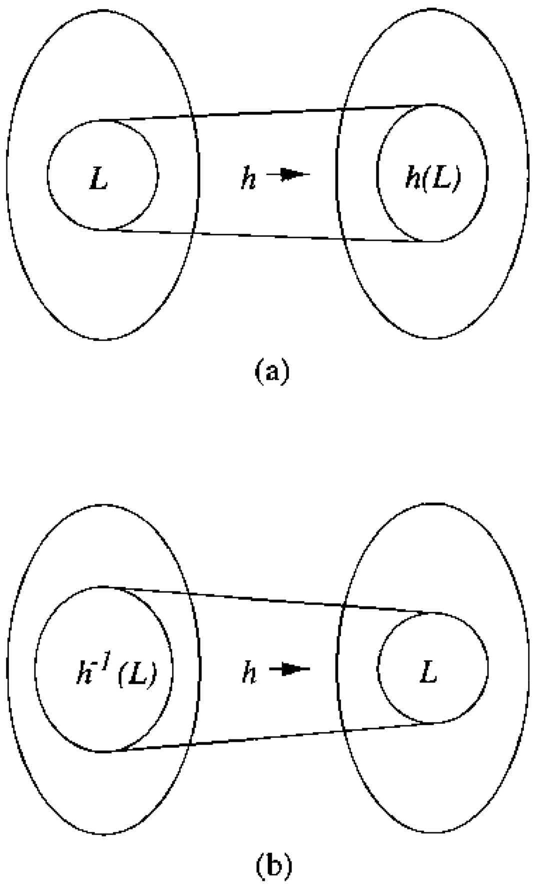


Figure 4.5: A homomorphism applied in the forward and inverse direction

(Only-if) Now, we must assume that $h(w)$ is in L and show that w is of the form $baba \cdots ba$. There are four conditions under which a string is *not* of that form, and we shall show that if any of them hold then $h(w)$ is not in L. That is, we prove the contrapositive of the statement we set out to prove.

1. If w begins with a, then $h(w)$ begins with 01. It therefore has an isolated 0, and is not in L.

2. If w ends in b, then $h(w)$ ends in 10, and again there is an isolated 0 in $h(w)$.

3. If w has two consecutive a's, then $h(w)$ has a substring 0101. Here too, there is an isolated 0 in w.

4. Likewise, if w has two consecutive b's, then $h(w)$ has substring 1010 and has an isolated 0.

Thus, whenever one of the above cases hold, $h(w)$ is not in L. However, unless at least one of items (1) through (4) hold, then w is of the form $baba \cdots ba$.

To see why, assume none of (1) through (4) hold. Then (1) tells us w must begin with b, and (2) tells us w ends with a. Statements (3) and (4) tell us that a's and b's must alternate in w. Thus, the logical "OR" of (1) through (4) is equivalent to the statement "w is not of the form $baba \cdots ba$." We have proved that the "OR" of (1) through (4) implies $h(w)$ is not in L. That statement is the contrapositive of the statement we wanted: "if $h(w)$ is in L, then w is of the form $baba \cdots ba$." $\square$

We shall next prove that the inverse homomorphism of a regular language is also regular, and then show how the theorem can be used.

Theorem 4.16: If h is a homomorphism from alphabet Σ to alphabet T, and L is a regular language over T, then $h^{-1}(L)$ is also a regular language.

PROOF: The proof starts with a DFA A for L. We construct from A and h a DFA for $h^{-1}(L)$ using the plan suggested by Fig. 4.6. This DFA uses the states of A but translates the input symbol according to h before deciding on the next state.

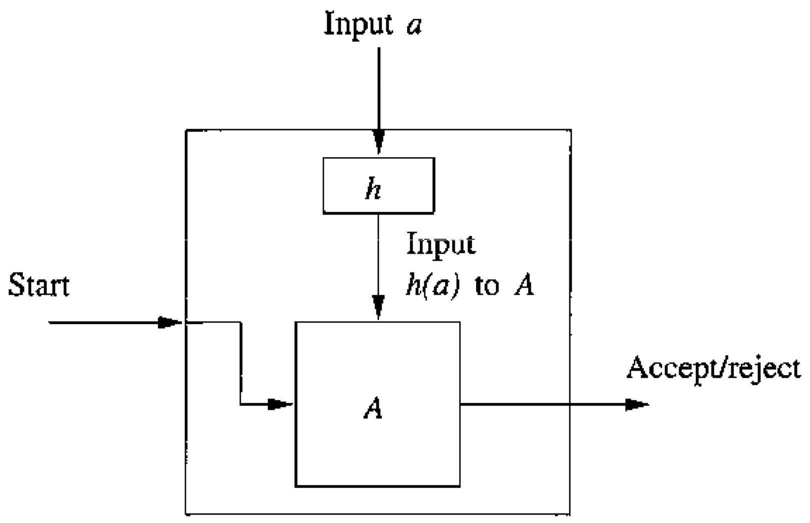


Figure 4.6: The DFA for $h^{-1}(L)$ applies h to its input, and then simulates the DFA for L

Formally, let L be $L(A)$, where DFA $A = (Q, T, \delta, q_0, F)$. Define a DFA

$$B = (Q, \Sigma, \gamma, q_0, F)$$

where transition function γ is constructed by the rule $\gamma(q, a) = \hat{\delta}\big(q, h(a)\big)$. That is, the transition B makes on input a is the result of the sequence of transitions that A makes on the string of symbols $h(a)$. Remember that $h(a)$ could be ϵ, it could be one symbol, or it could be *many* symbols, but $\hat{\delta}$ is properly defined to take care of all these cases.

It is an easy induction on $|w|$ to show that $\hat{\gamma}(q_0, w) = \hat{\delta}\big(q_0, h(w)\big)$. Since the accepting states of A and B are the same, B accepts w if and only if A accepts $h(w)$. Put another way, B accepts exactly those strings w that are in $h^{-1}(L)$. $\square$

Example 4.17: In this example we shall use inverse homomorphism and several other closure properties of regular sets to prove an odd fact about finite automata. Suppose we required that a DFA visit every state at least once when accepting its input. More precisely, suppose $A = (Q, \Sigma, \delta, q_0, F)$ is a DFA, and we are interested in the language L of all strings w in Σ^* such that $\hat{\delta}(q_0, w)$ is in F, and also for every state q in Q there is some prefix x_q of w such that $\hat{\delta}(q_0, x_q) = q$. Is L regular? We can show it is, but the construction is complex.

First, start with the language M that is $L(A)$, i.e., the set of strings that A accepts in the usual way, without regard to what states it visits during the processing of its input. Note that $L \subseteq M$, since the definition of L puts an additional condition on the strings of $L(A)$. Our proof that L is regular begins by using an inverse homomorphism to, in effect, place the states of A into the input symbols. More precisely, let us define a new alphabet T consisting of symbols that we may think of as triples $[paq]$, where:

1. p and q are states in Q,

2. a is a symbol in Σ, and

3. $\delta(p, a) = q$.

That is, we may think of the symbols in T as representing transitions of the automaton A. It is important to see that the notation $[paq]$ is our way of expressing a single symbol, not the concatenation of three symbols. We could have given it a single letter as a name, but then its relationship to p, q, and a would be hard to describe.

Now, define the homomorphism $h([paq]) = a$ for all p, a, and q. That is, h removes the state components from each of the symbols of T and leaves only the symbol from Σ. Our first step in showing L is regular is to construct the language $L_1 = h^{-1}(M)$. Since M is regular, so is L_1 by Theorem 4.16. The strings of L_1 are just the strings of M with a pair of states, representing a transition, attached to each symbol.

As a very simple illustration, consider the two-state automaton of Fig. 4.4(a). The alphabet Σ is $\{0, 1\}$, and the alphabet T consists of the four symbols $[p0q]$, $[q0q]$, $[p1p]$, and $[q1q]$. For instance, there is a transition from state p to q on input 0, so $[p0q]$ is one of the symbols of T. Since 101 is a string accepted by the automaton, h^{-1} applied to this string will give us $2^3 = 8$ strings, of which $[p1p][p0q][q1q]$ and $[q1q][q0q][p1p]$ are two examples.

We shall now construct L from L_1 by using a series of further operations that preserve regular languages. Our first goal is to eliminate all those strings of L_1 that deal incorrectly with states. That is, we can think of a symbol like

$[paq]$ as saying the automaton was in state p, read input a, and thus entered state q. The sequence of symbols must satisfy three conditions if it is to be deemed an accepting computation of A:

1. The first state in the first symbol must be q_0, the start state of A.
2. Each transition must pick up where the previous one left off. That is, the first state in one symbol must equal the second state of the previous symbol.
3. The second state of the last symbol must be in F. This condition in fact will be guaranteed once we enforce (1) and (2), since we know that every string in L_1 came from a string accepted by A.

The plan of the construction of L is shown in Fig. 4.7.

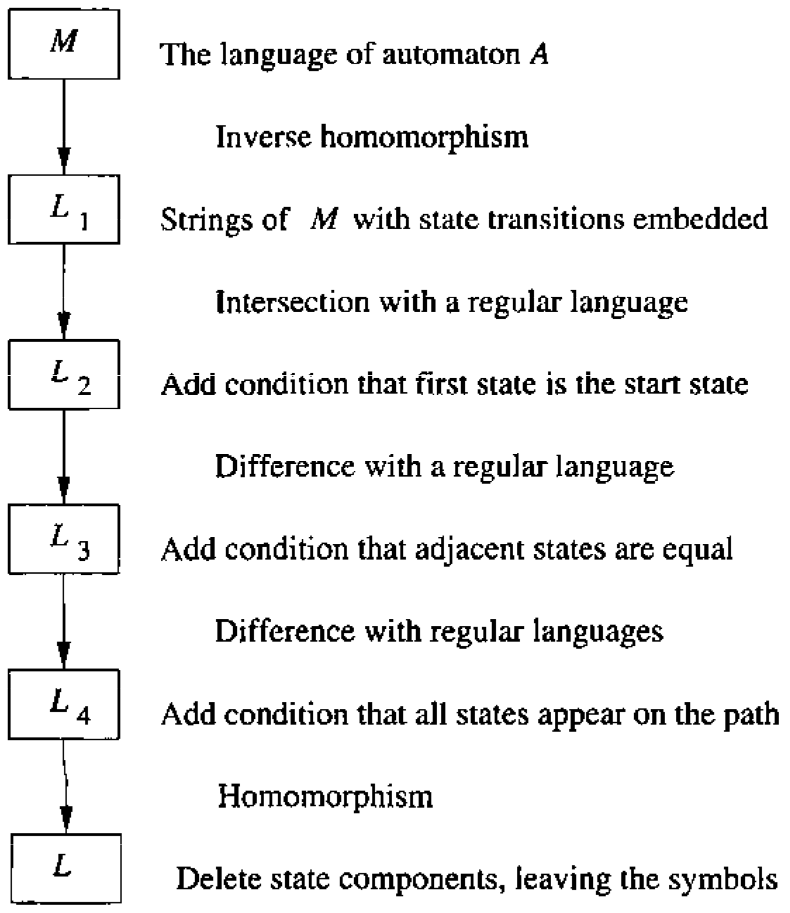

M
The language of automaton A
Inverse homomorphism
L 1
Strings of M with state transitions embedded
Intersection with a regular language
L 2
Add condition that first state is the start state
Difference with a regular language
L 3
Add condition that adjacent states are equal
Difference with regular languages
L 4
Add condition that all states appear on the path
Homomorphism
L
Delete state components, leaving the symbols

Figure 4.7: Constructing language L from language M by applying operations that preserve regularity of languages

We enforce (1) by intersecting L_1 with the set of strings that begin with a symbol of the form $[q_0aq]$ for some symbol a and state q. That is, let E_1 be the expression $[q_0a_1q_1] + [q_0a_2q_2] + \cdots$, where the pairs a_iq_i range over all pairs in $\Sigma \times Q$ such that $\delta(q_0, a_i) = q_i$. Then let $L_2 = L_1 \cap L(E_1T^*)$. Since E_1T^* is a regular expression denoting all strings in T^* that begin with the start state (treat T in the regular expression as the sum of its symbols), L_2 is all strings that are formed by applying h^{-1} to language M and that have the start state as the first component of its first symbol; i.e., it meets condition (1).

To enforce condition (2), it is easier to subtract from L_2 (using the set-difference operation) all those strings that violate it. Let E_2 be the regular expression consisting of the sum (union) of the concatenation of all pairs of symbols that fail to match; that is, pairs of the form $[paq][rbs]$ where $q \neq r$. Then $T^*E_2T^*$ is a regular expression denoting all strings that fail to meet condition (2).

We may now define $L_3 = L_2 - L(T^*E_2T^*)$. The strings of L_3 satisfy condition (1) because strings in L_2 must begin with the start symbol. They satisfy condition (2) because the subtraction of $L(T^*E_2T^*)$ removes any string that violates that condition. Finally, they satisfy condition (3), that the last state is accepting, because we started with only strings in M, all of which lead to acceptance by A. The effect is that L_3 consists of the strings in M with the states of the accepting computation of that string embedded as part of each symbol. Note that L_3 is regular because it is the result of starting with the regular language M, and applying operations — inverse homomorphism, intersection, and set difference — that yield regular sets when applied to regular sets.

Recall that our goal was to accept only those strings in M that visited every state in their accepting computation. We may enforce this condition by additional applications of the set-difference operator. That is, for each state q, let E_q be the regular expression that is the sum of all the symbols in T such that q appears in neither its first or last position. If we subtract $L(E_q^*)$ from L_3 we have those strings that are an accepting computation of A and that visit state q at least once. If we subtract from L_3 all the languages $L(E_q^*)$ for q in Q, then we have the accepting computations of A that visit all the states. Call this language L_4. By Theorem 4.10 we know L_4 is also regular.

Our final step is to construct L from L_4 by getting rid of the state components. That is, $L = h(L_4)$. Now, L is the set of strings in Σ^* that are accepted by A and that visit each state of A at least once during their acceptance. Since regular languages are closed under homomorphisms, we conclude that L is regular. □

4.2.5 Exercises for Section 4.2

Exercise 4.2.1: Suppose h is the homomorphism from the alphabet $\{0, 1, 2\}$ to the alphabet $\{a, b\}$ defined by: $h(0) = a$; $h(1) = ab$, and $h(2) = ba$.

* a) What is $h(0120)$?

b) What is $h(21120)$?

* c) If L is the language $L(\mathbf{01^*2})$, what is $h(L)$?

d) If L is the language $L(\mathbf{0} + \mathbf{12})$, what is $h(L)$?

* e) Suppose L is the language $\{ababa\}$, that is, the language consisting of only the one string $ababa$. What is $h^{-1}(L)$?

! f) If L is the language $L\big(\mathsf{a}(\mathsf{ba})^*\big)$, what is $h^{-1}(L)$?

*! **Exercise 4.2.2:** If L is a language, and a is a symbol, then L/a, the *quotient* of L and a, is the set of strings w such that wa is in L. For example, if $L = \{a, aab, baa\}$, then $L/a = \{\epsilon, ba\}$. Prove that if L is regular, so is L/a. *Hint*: Start with a DFA for L and consider the set of accepting states.

! **Exercise 4.2.3:** Lf L is a language, and a is a symbol, then $a\backslash L$ is the set of strings w such that aw is in L. For example, if $L = \{a, aab, baa\}$, then $a\backslash L = \{\epsilon, ab\}$. Prove that if L is regular, so is $a\backslash L$. *Hint*: Remember that the regular languages are closed under reversal and under the quotient operation of Exercise 4.2.2.

! **Exercise 4.2.4:** Which of the following identities are true?

a) $(L/a)a = L$ (the left side represents the concatenation of the languages L/a and $\{a\}$).

b) $a(a\backslash L) = L$ (again, concatenation with $\{a\}$, this time on the left, is intended).

c) $(La)/a = L$.

d) $a\backslash(aL) = L$.

Exercise 4.2.5: The operation of Exercise 4.2.3 is sometimes viewed as a "derivative," and $a\backslash L$ is written $\frac{dL}{da}$. These derivatives apply to regular expressions in a manner similar to the way ordinary derivatives apply to arithmetic expressions. Thus, if R is a regular expression, we shall use $\frac{dR}{da}$ to mean the same as $\frac{dL}{da}$, if $L = L(R)$.

a) Show that $\frac{d(R+S)}{da} = \frac{dR}{da} + \frac{dS}{da}$.

*! b) Give the rule for the "derivative" of RS. *Hint*: You need to consider two cases: if $L(R)$ does or does not contain ϵ. This rule is not quite the same as the "product rule" for ordinary derivatives, but is similar.

! c) Give the rule for the "derivative" of a closure, i.e., $\frac{d(R^*)}{da}$.

d) Use the rules from (a)-(c) to find the "derivatives" of regular expression $(\mathbf{0} + \mathbf{1})^*\mathbf{011}$ with respect to 0 and 1.

* e) Characterize those languages L for which $\frac{dL}{d0} = \emptyset$.

*! f) Characterize those languages L for which $\frac{dL}{d0} = L$.

! **Exercise 4.2.6:** Show that the regular languages are closed under the following operations:

a) $\mathit{min}(L) = \{w \mid w$ is in L, but no proper prefix of w is in $L\}$.

b) $\mathit{max}(L) = \{w \mid w$ is in L and for no x other than ϵ is wx in $L\}$.

c) $\mathit{init}(L) = \{w \mid$ for some x, wx is in $L\}$.

Hint: Like Exercise 4.2.2, it is easiest to start with a DFA for L and perform a construction to get the desired language.

! **Exercise 4.2.7:** If $w = a_1a_2\cdots a_n$ and $x = b_1b_2\cdots b_m$ are strings of the same length, define $\mathit{alt}(w, x)$ to be the string in which the symbols of w and x alternate, starting with w, that is, $a_1b_1a_2b_2\cdots a_nb_n$. If L and M are languages, define $\mathit{alt}(L, M)$ to be the set of strings of the form $\mathit{alt}(w, x)$, where w is any string in L and x is any string in M of the same length. Prove that if L and M are regular, so is $\mathit{alt}(L, M)$.

*!! **Exercise 4.2.8:** Let L be a language. Define $\mathit{half}(L)$ to be the set of first halves of strings in L, that is, $\{w \mid$ for some x such that $|x| = |w|$, we have wx in $L\}$. For example, if $L = \{\epsilon, 0010, 011, 010110\}$ then $\mathit{half}(L) = \{\epsilon, 00, 010\}$. Notice that odd-length strings do not contribute to $\mathit{half}(L)$. Prove that if L is a regular language, so is $\mathit{half}(L)$.

!! **Exercise 4.2.9:** We can generalize Exercise 4.2.8 to a number of functions that determine how much of the string we take. If f is a function of integers, define $f(L)$ to be $\{w \mid$ for some x, with $|x| = f(|w|)$, we have wx in $L\}$. For instance, the operation *half* corresponds to f being the identity function $f(n) = n$, since $\mathit{half}(L)$ is defined by having $|x| = |w|$. Show that if L is a regular language, then so is $f(L)$, if f is one of the following functions:

a) $f(n) = 2n$ (i.e., take the first thirds of strings).

b) $f(n) = n^2$ (i.e., the amount we take has length equal to the square root of what we do not take.

c) $f(n) = 2^n$ (i.e., what we take has length equal to the logarithm of what we leave).

!! **Exercise 4.2.10:** Suppose that L is any language, not necessarily regular, whose alphabet is $\{0\}$; i.e., the strings of L consist of 0's only. Prove that L^* is regular. *Hint*: At first, this theorem sounds preposterous. However, an example will help you see why it is true. Consider the language $L = \{0^i \mid i$ is prime$\}$, which we know is not regular by Example 4.3. Strings 00 and 000 are in L, since 2 and 3 are both primes. Thus, if $j \geq 2$, we can show 0^j is in L^*. If j is even, use $j/2$ copies of 00, and if j is odd, use one copy of 000 and $(j-3)/2$ copies of 00. Thus, $L^* = \mathbf{000^*}$.

!! **Exercise 4.2.11:** Show that the regular languages are closed under the following operation: $cycle(L) = \{w \mid \text{we can write } w \text{ as } w = xy, \text{ such that } yx \text{ is in } L\}$. For example, if $L = \{01, 011\}$, then $cycle(L) = \{01, 10, 011, 110, 101\}$. *Hint*: Start with a DFA for L and construct an ϵ-NFA for $cycle(L)$.

!! **Exercise 4.2.12:** Let $w_1 = a_0a_0a_1$, and $w_i = w_{i-1}w_{i-1}a_i$ for all $i > 1$. For instance, $w_3 = a_0a_0a_1a_0a_0a_1a_2a_0a_0a_1a_0a_0a_1a_2a_3$. The shortest regular expression for the language $L_n = \{w_n\}$, i.e., the language consisting of the one string w_n, is the string w_n itself, and the length of this expression is $2^{n+1} - 1$. However, if we allow the intersection operator, we can write an expression for L_n whose length is $O(n^2)$. Find such an expression. *Hint*: Find n languages, each with regular expressions of length $O(n)$, whose intersection is L_n.

! **Exercise 4.2.13:** We can use closure properties to help prove certain languages are not regular. Start with the fact that the language

$$L_{0^n1^n} = \{0^n1^n \mid n \geq 0\}$$

is not a regular set. Prove the following languages not to be regular by transforming them, using operations known to preserve regularity, to $L_{0^n1^n}$:

* a) $\{0^i1^j \mid i \neq j\}$.

b) $\{0^n1^m2^{n-m} \mid n \geq m \geq 0\}$.

Exercise 4.2.14: In Theorem 4.8, we described the "product construction" that took two DFA's and constructed one DFA whose language is the intersection of the languages of the first two.

a) Show how to perform the product construction on NFA's (without ϵ-transitions).

! b) Show how to perform the product construction on ϵ-NFA's.

* c) Show how to modify the product construction so the resulting DFA accepts the difference of the languages of the two given DFA's.

d) Show how to modify the product construction so the resulting DFA accepts the union of the languages of the two given DFA's.

Exercise 4.2.15: In the proof of Theorem 4.8 we claimed that it could be proved by induction on the length of w that

$$\hat{\delta}\big((q_L, q_M), w\big) = \big(\hat{\delta}_L(q_L, w), \hat{\delta}_M(q_M, w)\big)$$

Give this inductive proof.

Exercise 4.2.16: Complete the proof of Theorem 4.14 by considering the cases where expression E is a concatenation of two subexpressions and where E is the closure of an expression.

Exercise 4.2.17: In Theorem 4.16, we omitted a proof by induction on the length of w that $\hat{\gamma}(q_0, w) = \hat{\delta}\big(q_0, h(w)\big)$. Prove this statement.

4.3 Decision Properties of Regular Languages

In this section we consider how one answers important questions about regular languages. First, we must consider what it means to ask a question about a language. The typical language is infinite, so you cannot present the strings of the language to someone and ask a question that requires them to inspect the infinite set of strings. Rather, we present a language by giving one of the finite representations for it that we have developed: a DFA, an NFA, an ϵ-NFA, or a regular expression.

Of course the language so described will be regular, and in fact there is no way at all to represent completely arbitrary languages. In later chapters we shall see finite ways to represent more than the regular languages, so we can consider questions about languages in these more general classes. However, for many of the questions we ask, algorithms exist only for the class of regular languages. The same questions become "undecidable" (no algorithm to answer them exists) when posed using more "expressive" notations (i.e., notations that can be used to express a larger set of languages) than the representations we have developed for the regular languages.

We begin our study of algorithms for questions about regular languages by reviewing the ways we can convert one representation into another for the same language. In particular, we want to observe the time complexity of the algorithms that perform the conversions. We then consider some of the fundamental questions about languages:

1. Is the language described empty?

2. Is a particular string w in the described language?

3. Do two descriptions of a language actually describe the same language? This question is often called "equivalence" of languages.

4.3.1 Converting Among Representations

We know that we can convert any of the four representations for regular languages to any of the other three representations. Figure 3.1 gave paths from any representation to any of the others. While there are algorithms for any of the conversions, sometimes we are interested not only in the possibility of making a conversion, but in the amount of time it takes. In particular, it is important to distinguish between algorithms that take exponential time (as a function of the size of their input), and therefore can be performed only for relatively small instances, from those that take time that is a linear, quadratic, or some small-degree polynomial of their input size. The latter algorithms are "realistic," in the sense that we expect them to be executable for large instances of the problem. We shall consider the time complexity of each of the conversions we discussed.

Converting NFA's to DFA's

When we start with either an NFA or and ϵ-NFA and convert it to a DFA, the time can be exponential in the number of states of the NFA. First, computing the ϵ-closure of n states takes $O(n^3)$ time. We must search from each of the n states along all arcs labeled ϵ. If there are n states, there can be no more than n^2 arcs. Judicious bookkeeping and well-designed data structures will make sure that we can explore from each state in $O(n^2)$ time. In fact, a transitive closure algorithm such as Warshall's algorithm can be used to compute the entire ϵ-closure at once.[3]

Once the ϵ-closure is computed, we can compute the equivalent DFA by the subset construction. The dominant cost is, in principle, the number of states of the DFA, which can be 2^n. For each state, we can compute the transitions in $O(n^3)$ time by consulting the ϵ-closure information and the NFA's transition table for each of the input symbols. That is, suppose we want to compute $\delta(\{q_1, q_2, \ldots, q_k\}, a)$ for the DFA. There may be as many as n states reachable from each q_i along ϵ-labeled paths, and each of those states may have up to n arcs labeled a. By creating an array indexed by states, we can compute the union of up to n sets of up to n states in time proportional to n^2.

In this way, we can compute, for each q_i, the set of states reachable from q_i along a path labeled a (possibly including ϵ's). Since $k \leq n$, there are at most n states to deal with. We compute the reachable states for each in $O(n^2)$ time. Thus, the total time spent computing reachable states is $O(n^3)$. The union of the sets of reachable states requires only $O(n^2)$ additional time, and we conclude that the computation of one DFA transition takes $O(n^3)$ time.

Note that the number of input symbols is assumed constant, and does not depend on n. Thus, in this and other estimates of running time, we do not consider the number of input symbols as a factor. The size of the input alphabet influences the constant factor that is hidden in the "big-oh" notation, but nothing more.

Our conclusion is that the running time of NFA-to-DFA conversion, including the case where the NFA has ϵ-transitions, is $O(n^3 2^n)$. Of course in practice it is common that the number of states created is much less than 2^n, often only n states. We could state the bound on the running time as $O(n^3 s)$, where s is the number of states the DFA actually has.

DFA-to-NFA Conversion

This conversion is simple, and takes $O(n)$ time on an n-state DFA. All that we need to do is modify the transition table for the DFA by putting set-brackets around states and, if the output is an ϵ-NFA, adding a column for ϵ. Since we treat the number of input symbols (i.e., the width of the transition table) as a constant, copying and processing the table takes $O(n)$ time.

[3]For a discussion of transitive closure algorithms, see A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1984.

Automaton-to-Regular-Expression Conversion

If we examine the construction of Section 3.2.1 we observe that at each of n rounds (where n is the number of states of the DFA) we can quadruple the size of the regular expressions constructed, since each is built from four expressions of the previous round. Thus, simply writing down the n^3 expressions can take time $O(n^3 4^n)$. The improved construction of Section 3.2.2 reduces the constant factor, but does not affect the worst-case exponentiality of the problem.

The same construction works in the same running time if the input is an NFA, or even an ϵ-NFA, although we did not prove those facts. It is important to use those constructions for NFA's, however. If we first convert an NFA to a DFA and then convert the DFA to a regular expression, it could take time $O(n^3 4^{n^3 2^n})$, which is doubly exponential.

Regular-Expression-to-Automaton Conversion

Conversion of a regular expression to an ϵ-NFA takes linear time. We need to parse the expression efficiently, using a technique that takes only $O(n)$ time on a regular expression of length n.[4] The result is an expression tree with one node for each symbol of the regular expression (although parentheses do not have to appear in the tree; they just guide the parsing of the expression).

Once we have an expression tree for the regular expression, we can work up the tree, building the ϵ-NFA for each node. The construction rules for the conversion of a regular expression that we saw in Section 3.2.3 never add more than two states and four arcs for any node of the expression tree. Thus, the numbers of states and arcs of the resulting ϵ-NFA are both $O(n)$. Moreover, the work at each node of the parse tree in creating these elements is constant, provided the function that processes each subtree returns pointers to the start and accepting states of its automaton.

We conclude that construction of an ϵ-NFA from a regular expression takes time that is linear in the size of the expression. We can eliminate ϵ-transitions from an n-state ϵ-NFA, to make an ordinary NFA, in $O(n^3)$ time, without increasing the number of states. However, proceeding to a DFA can take exponential time.

4.3.2 Testing Emptiness of Regular Languages

At first glance the answer to the question "is regular language L empty?" is obvious: $\emptyset$ is empty, and all other regular languages are not. However, as we discussed at the beginning of Section 4.3, the problem is not stated with an explicit list of the strings in L. Rather, we are given some representation for L and need to decide whether that representation denotes the language $\emptyset$.

[4]Parsing methods capable of doing this task in $O(n)$ time are discussed in A. V. Aho, R. Sethi, and J. D. Ullman, *Compiler Design: Principles, Tools, and Techniques*, Addison-Wesley, 1986.

If our representation is any kind of finite automaton, the emptiness question is whether there is any path whatsoever from the start state to some accepting state. If so, the language is nonempty, while if the accepting states are all separated from the start state, then the language is empty. Deciding whether we can reach an accepting state from the start state is a simple instance of graph-reachability, similar in spirit to the calculation of the ϵ-closure that we discussed in Section 2.5.3. The algorithm can be summarized by this recursive process.

BASIS: The start state is surely reachable from the start state.

INDUCTION: If state q is reachable from the start state, and there is an arc from q to p with any label (an input symbol, or ϵ if the automaton is an ϵ-NFA), then p is reachable.

In that manner we can compute the set of reachable states. If any accepting state is among them, we answer "no" (the language of the automaton is *not* empty), and otherwise we answer "yes." Note that the reachability calculation takes no more time that $O(n^2)$ if the automaton has n states, and in fact it is no worse than proportional to the number of arcs in the automaton's transition diagram, which could be less than n^2 and cannot be more than $O(n^2)$.

If we are given a regular expression representing the language L, rather than an automaton, we could convert the expression to an ϵ-NFA and proceed as above. Since the automaton that results from a regular expression of length n has at most $O(n)$ states and transitions, the algorithm takes $O(n)$ time.

However, we can also inspect the regular expression to decide whether it is empty. Notice first that if the expression has no occurrence of $\emptyset$, then its language is surely not empty. If there are $\emptyset$'s, the language may or may not be empty. The following recursive rules tell whether a regular expression denotes the empty language.

BASIS: $\emptyset$ denotes the empty language; ϵ and $\mathbf{a}$ for any input symbol a do not.

INDUCTION: Suppose R is a regular expression. There are four cases to consider, corresponding to the ways that R could be constructed.

1. $R = R_1 + R_2$. Then $L(R)$ is empty if and only if both $L(R_1)$ and $L(R_2)$ are empty.

2. $R = R_1R_2$. Then $L(R)$ is empty if and only if either $L(R_1)$ or $L(R_2)$ is empty.

3. $R = R_1^*$. Then $L(R)$ is not empty; it always includes at least ϵ.

4. $R = (R_1)$. Then $L(R)$ is empty if and only if $L(R_1)$ is empty, since they are the same language.

4.3.3 Testing Membership in a Regular Language

The next question of importance is, given a string w and a regular language L, is w in L. While w is represented explicitly, L is represented by an automaton or regular expression.

If L is represented by a DFA, the algorithm is simple. Simulate the DFA processing the string of input symbols w, beginning in the start state. If the DFA ends in an accepting state, the answer is "yes"; otherwise the answer is "no." This algorithm is extremely fast. If $|w| = n$, and the DFA is represented by a suitable data structure, such as a two-dimensional array that is the transition table, then each transition requires constant time, and the entire test takes $O(n)$ time.

If L has any other representation besides a DFA, we could convert to a DFA and run the test above. That approach could take time that is exponential in the size of the representation, although it is linear in $|w|$. However, if the representation is an NFA or ϵ-NFA, it is simpler and more efficient to simulate the NFA directly. That is, we process symbols of w one at a time, maintaining the set of states the NFA can be in after following any path labeled with that prefix of w. The idea was presented in Fig. 2.10.

If w is of length n, and the NFA has s states, then the running time of this algorithm is $O(ns^2)$. Each input symbol can be processed by taking the previous set of states, which numbers at most s states, and looking at the successors of each of these states. We take the union of at most s sets of at most s states each, which requires $O(s^2)$ time.

If the NFA has ϵ-transitions, then we must compute the ϵ-closure before starting the simulation. Then the processing of each input symbol a has two stages, each of which requires $O(s^2)$ time. First, we take the previous set of states and find their successors on input symbol a. Next, we compute the ϵ-closure of this set of states. The initial set of states for the simulation is the ϵ-closure of the initial state of the NFA.

Lastly, if the representation of L is a regular expression of size s, we can convert to an ϵ-NFA with at most $2s$ states, in $O(s)$ time. We then perform the simulation above, taking $O(ns^2)$ time on an input w of length n.

4.3.4 Exercises for Section 4.3

* **Exercise 4.3.1:** Give an algorithm to tell whether a regular language L is infinite. *Hint*: Use the pumping lemma to show that if the language contains any string whose length is above a certain lower limit, then the language must be infinite.

Exercise 4.3.2: Give an algorithm to tell whether a regular language L contains at least 100 strings.

Exercise 4.3.3: Suppose L is a regular language with alphabet Σ. Give an algorithm to tell whether $L = \Sigma^*$, i.e., all strings over its alphabet.

Exercise 4.3.4: Give an algorithm to tell whether two regular languages L_1 and L_2 have at least one string in common.

Exercise 4.3.5: Give an algorithm to tell, for two regular languages L_1 and L_2 over the same alphabet Σ, whether there is any string in Σ^* that is in neither L_1 nor L_2.

4.4 Equivalence and Minimization of Automata

In contrast to the previous questions — emptiness and membership — whose algorithms were rather simple, the question of whether two descriptions of two regular languages actually define the same language involves considerable intellectual mechanics. In this section we discuss how to test whether two descriptors for regular languages are ***equivalent***, in the sense that they define the same language. An important consequence of this test is that there is a way to minimize a DFA. That is, we can take any DFA and find an equivalent DFA that has the minimum number of states. In fact, this DFA is essentially unique: given any two minimum-state DFA's that are equivalent, we can always find a way to rename the states so that the two DFA's become the same.

4.4.1 Testing Equivalence of States

We shall begin by asking a question about the states of a single DFA. Our goal is to understand when two distinct states p and q can be replaced by a single state that behaves like both p and q. We say that states p and q are ***equivalent*** if:

- For all input strings w, $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is an accepting state.

Less formally, it is impossible to tell the difference between equivalent states p and q merely by starting in one of the states and asking whether or not a given input string leads to acceptance when the automaton is started in this (unknown) state. Note we do *not* require that $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ are the *same* state, only that either both are accepting or both are nonaccepting.

If two states are not equivalent, then we say they are ***distinguishable***. That is, state p is distinguishable from state q if there is at least one string w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and the other is not accepting.

Example 4.18: Consider the DFA of Fig. 4.8, whose transition function we shall refer to as δ in this example. Certain pairs of states are obviously not equivalent. For example, C and G are not equivalent because one is accepting and the other is not. That is, the empty string distinguishes these two states, because $\hat{\delta}(C, \epsilon)$ is accepting and $\hat{\delta}(G, \epsilon)$ is not.

Consider states A and G. String ϵ doesn't distinguish them, because they are both nonaccepting states. String 0 doesn't distinguish them because they go to

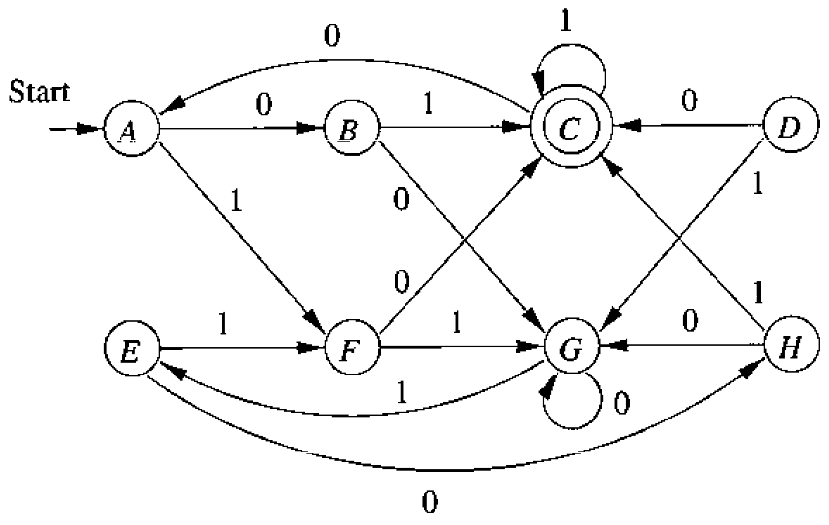


Figure 4.8: An automaton with equivalent states

states B and G, respectively on input 0, and both these states are nonaccepting. Likewise, string 1 doesn't distinguish A from G, because they go to F and E, respectively, and both are nonaccepting. However, 01 distinguishes A from G, because $\hat{\delta}(A, 01) = C$, $\hat{\delta}(G, 01) = E$, C is accepting, and E is not. Any input string that takes A and G to states only one of which is accepting is sufficient to prove that A and G are not equivalent.

In contrast, consider states A and E. Neither is accepting, so ϵ does not distinguish them. On input 1, they both go to state F. Thus, no input string that begins with 1 can distinguish A from E, since for any string x, $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x)$.

Now consider the behavior of states A and E on inputs that begin with 0. They go to states B and H, respectively. Since neither is accepting, string 0 by itself does not distinguish A from E. However, B and H are no help. On input 1 they both go to C, and on input 0 they both go to G. Thus, all inputs that begin with 0 will fail to distinguish A from E. We conclude that no input string whatsoever will distinguish A from E; i.e., they are equivalent states. □

To find states that are equivalent, we make our best efforts to find pairs of states that are distinguishable. It is perhaps surprising, but true, that if we try our best, according to the algorithm to be described below, then any pair of states that we do not find distinguishable are equivalent. The algorithm, which we refer to as the ***table-filling algorithm***, is a recursive discovery of distinguishable pairs in a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

BASIS: If p is an accepting state and q is nonaccepting, then the pair $\{p, q\}$ is distinguishable.

INDUCTION: Let p and q be states such that for some input symbol a, $r = \delta(p, a)$ and $s = \delta(q, a)$ are a pair of states known to be distinguishable. Then

$\{p,q\}$ is a pair of distinguishable states. The reason this rule makes sense is that there must be some string w that distinguishes r from s; that is, exactly one of $\hat{\delta}(r,w)$ and $\hat{\delta}(s,w)$ is accepting. Then string aw must distinguish p from q, since $\hat{\delta}(p,aw)$ and $\hat{\delta}(q,aw)$ is the same pair of states as $\hat{\delta}(r,w)$ and $\hat{\delta}(s,w)$.

Example 4.19: Let us execute the table-filling algorithm on the DFA of Fig 4.8. The final table is shown in Fig. 4.9, where an x indicates pairs of distinguishable states, and the blank squares indicate those pairs that have been found equivalent. Initially, there are no x's in the table.

B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x
	A	B	C	D	E	F	G

Figure 4.9: Table of state inequivalences

For the basis, since C is the only accepting state, we put x's in each pair that involves C. Now that we know some distinguishable pairs, we can discover others. For instance, since $\{C,H\}$ is distinguishable, and states E and F go to H and C, respectively, on input 0, we know that $\{E,F\}$ is also a distinguishable pair. In fact, all the x's in Fig. 4.9 with the exception of the pair $\{A,G\}$ can be discovered simply by looking at the transitions from the pair of states on either 0 or on 1, and observing that (for one of those inputs) one state goes to C and the other does not. We can show $\{A,G\}$ is distinguishable on the next round, since on input 1 they go to F and E, respectively, and we already established that the pair $\{E,F\}$ is distinguishable.

However, then we can discover no more distinguishable pairs. The three remaining pairs, which are therefore equivalent pairs, are $\{A,E\}$, $\{B,H\}$, and $\{D,F\}$. For example, consider why we can not infer that $\{A,E\}$ is a distinguishable pair. On input 0, A and E go to B and H, respectively, and $\{B,H\}$ has not yet been shown distinguishable. On input 1, A and E both go to F, so there is no hope of distinguishing them that way. The other two pairs, $\{B,H\}$ and $\{D,F\}$ will never be distinguished because they each have identical transitions on 0 and identical transitions on 1. Thus, the table-filling algorithm stops with the table as shown in Fig. 4.9, which is the correct determination of equivalent and distinguishable states. □

Theorem 4.20: If two states are not distinguished by the table-filling algorithm, then the states are equivalent.

PROOF: Let us again assume we are talking of the DFA $A = (Q, \Sigma, \delta, q_0, F)$. Suppose the theorem is false; that is, there is at least one pair of states $\{p, q\}$ such that

1. States p and q are distinguishable, in the sense that there is some string w such that exactly one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and yet

2. The table-filling algorithm does not find p and q to be distinguished.

Call such a pair of states a *bad pair*.

If there are bad pairs, then there must be some that are distinguished by the shortest strings among all those strings that distinguish bad pairs. Let $\{p, q\}$ be one such bad pair, and let $w = a_1a_2 \cdots a_n$ be a string as short as any that distinguishes p from q. Then exactly one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting.

Observe first that w cannot be ϵ, since if ϵ distinguishes a pair of states, then that pair is marked by the basis part of the table-filling algorithm. Thus, $n \geq 1$.

Consider the states $r = \delta(p, a_1)$ and $s = \delta(q, a_1)$. States r and s are distinguished by the string $a_2a_3 \cdots a_n$, since this string takes r and s to the states $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$. However, the string distinguishing r from s is shorter than any string that distinguishes a bad pair. Thus, $\{r, s\}$ cannot be a bad pair. Rather, the table-filling algorithm must have discovered that they are distinguishable.

But the inductive part of the table-filling algorithm will not stop until it has also inferred that p and q are distinguishable, since it finds that $\delta(p, a_1) = r$ is distinguishable from $\delta(q, a_1) = s$. We have contradicted our assumption that bad pairs exist. If there are no bad pairs, then every pair of distinguishable states is distinguished by the table-filling algorithm, and the theorem is true. □

4.4.2 Testing Equivalence of Regular Languages

The table-filling algorithm gives us an easy way to test if two regular languages are the same. Suppose languages L and M are each represented in some way, e.g., one by a regular expression and one by an NFA. Convert each representation to a DFA. Now, imagine one DFA whose states are the union of the states of the DFA's for L and M. Technically, this DFA has two start states, but actually the start state is irrelevant as far as testing state equivalence is concerned, so make any state the lone start state.

Now, test if the start states of the two original DFA's are equivalent, using the table-filling algorithm. If they are equivalent, then $L = M$, and if not, then $L \neq M$.

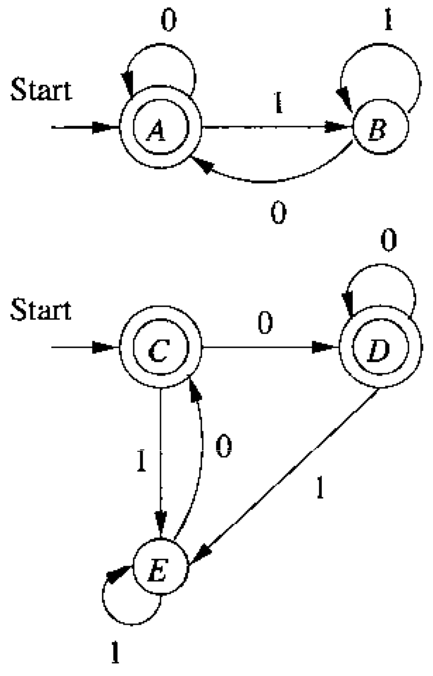


Figure 4.10: Two equivalent DFA's

Example 4.21: Consider the two DFA's in Fig. 4.10. Each DFA accepts the empty string and all strings that end in 0; that is the language of regular expression $\epsilon + (\mathbf{0} + \mathbf{1})^*\mathbf{0}$. We can imagine that Fig. 4.10 represents a single DFA, with five states A through E. If we apply the table-filling algorithm to that automaton, the result is as shown in Fig. 4.11.

B	*x*			
C		*x*		
D		*x*		
E	*x*		*x*	*x*
	A	*B*	*C*	*D*

Figure 4.11: The table of distinguishabilities for Fig. 4.10

To see how the table is filled out, we start by placing x's in all pairs of states where exactly one of the states is accepting. It turns out that there is no more to do. The four remaining pairs, $\{A, C\}$, $\{A, D\}$, $\{C, D\}$, and $\{B, E\}$ are all equivalent pairs. You should check that no more distinguishable pairs are discovered in the inductive part of the table-filling algorithm. For instance, with the table as in Fig. 4.11, we cannot distinguish the pair $\{A, D\}$ because on 0 they go to themselves, and on 1 they go to the pair $\{B, E\}$, which has not yet been distinguished. Since A and C are found equivalent by this test, and those states were the start states of the two original automata, we conclude that these DFA's do accept the same language. □

The time to fill out the table, and thus to decide whether two states are equivalent is polynomial in the number of states. If there are n states, then there are $\binom{n}{2}$, or $n(n-1)/2$ pairs of states. In one round, we consider all pairs of states, to see if one of their successor pairs has been found distinguishable, so a round surely takes no more than $O(n^2)$ time. Moreover, if on some round, no additional x's are placed in the table, then the algorithm ends. Thus, there can be no more than $O(n^2)$ rounds, and $O(n^4)$ is surely an upper bound on the running time of the table-filling algorithm.

However, a more careful algorithm can fill the table in $O(n^2)$ time. The idea is to initialize, for each pair of states $\{r, s\}$, a list of those pairs $\{p, q\}$ that "depend on" $\{r, s\}$. That is, if $\{r, s\}$ is found distinguishable, then $\{p, q\}$ is distinguishable. We create the lists initially by examining each pair of states $\{p, q\}$, and for each of the fixed number of input symbols a, we put $\{p, q\}$ on the list for the pair of states $\{\delta(p, a), \delta(q, a)\}$, which are the successor states for p and q on input a.

If we ever find $\{r, s\}$ to be distinguishable, then we go down the list for $\{r, s\}$. For each pair on that list that is not already distinguishable, we make that pair distinguishable, and we put the pair on a queue of pairs whose lists we must check similarly.

The total work of this algorithm is proportional to the sum of the lengths of the lists, since we are at all times either adding something to the lists (initialization) or examining a member of the list for the first and last time (when we go down the list for a pair that has been found distinguishable). Since the size of the input alphabet is considered a constant, each pair of states is put on $O(1)$ lists. As there are $O(n^2)$ pairs, the total work is $O(n^2)$.

4.4.3 Minimization of DFA's

Another important consequence of the test for equivalence of states is that we can "minimize" DFA's. That is, for each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language. Moreover, except for our ability to call the states by whatever names we choose, this minimum-state DFA is unique for the language. The algorithm is as follows:

1. First, eliminate any state that cannot be reached from the start state.

2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent. Theorem 4.24, below, shows that we can always make such a partition.

Example 4.22: Consider the table of Fig. 4.9, where we determined the state equivalences and distinguishabilities for the states of Fig. 4.8. The partition of the states into equivalent blocks is $(\{A, E\}, \{B, H\}, \{C\}, \{D, F\}, \{G\})$. Notice that the three pairs of states that are equivalent are each placed in a block together, while the states that are distinguishable from all the other states are each in a block alone.

For the automaton of Fig. 4.10, the partition is $(\{A, C, D\}, \{B, E\})$. This example shows that we can have more than two states in a block. It may appear fortuitous that A, C, and D can all live together in a block, because every pair of them is equivalent, and none of them is equivalent to any other state. However, as we shall see in the next theorem to be proved, this situation is guaranteed by our definition of "equivalence" for states. □

Theorem 4.23: The equivalence of states is transitive. That is, if in some DFA $A = (Q, \Sigma, \delta, q_0, F)$ we find that states p and q are equivalent, and we also find that q and r are equivalent, then it must be that p and r are equivalent.

PROOF: Note that transitivity is a property we expect of any relationship called "equivalence." However, simply calling something "equivalence" doesn't make it transitive; we must prove that the name is justified.

Suppose that the pairs $\{p, q\}$ and $\{q, r\}$ are equivalent, but pair $\{p, r\}$ is distinguishable. Then there is some input string w such that exactly one of $\hat{\delta}(p, w)$ and $\hat{\delta}(r, w)$ is an accepting state. Suppose, by symmetry, that $\hat{\delta}(p, w)$ is the accepting state.

Now consider whether $\hat{\delta}(q, w)$ is accepting or not. If it is accepting, then $\{q, r\}$ is distinguishable, since $\hat{\delta}(q, w)$ is accepting, and $\hat{\delta}(r, w)$ is not. If $\hat{\delta}(q, w)$ is nonaccepting, then $\{p, q\}$ is distinguishable for a similar reason. We conclude by contradiction that $\{p, r\}$ was not distinguishable, and therefore this pair is equivalent. □

We can use Theorem 4.23 to justify the obvious algorithm for partitioning states. For each state q, construct a block that consists of q and all the states that are equivalent to q. We must show that the resulting blocks are a partition; that is, no state is in two distinct blocks.

First, observe that all states in any block are mutually equivalent. That is, if p and r are two states in the block of states equivalent to q, then p and r are equivalent to each other, by Theorem 4.23.

Suppose that there are two overlapping, but not identical blocks. That is, there is a block B that includes states p and q, and another block C that includes p but not q. Since p and q are in a block together, they are equivalent. Consider how the block C was formed. If it was the block generated by p, then q would be in C, because those states are equivalent. Thus, it must be that there is some third state s that generated block C; i.e., C is the set of states equivalent to s.

We know that p is equivalent to s, because p is in block C. We also know that p is equivalent to q because they are together in block B. By the transitivity of Theorem 4.23, q is equivalent to s. But then q belongs in block C, a contradiction. We conclude that equivalence of states partitions the states; that is, two states either have the same set of equivalent states (including themselves), or their equivalent states are disjoint. To conclude the above analysis:

Theorem 4.24: If we create for each state q of a DFA a *block* consisting of q and all the states equivalent to q, then the different blocks of states form a *partition* of the set of states.[5] That is, each state is in exactly one block. All members of a block are equivalent, and no pair of states chosen from different blocks are equivalent. □

We are now able to state succinctly the algorithm for minimizing a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

1. Use the table-filling algorithm to find all the pairs of equivalent states.
2. Partition the set of states Q into blocks of mutually equivalent states by the method described above.
3. Construct the minimum-state equivalent DFA B by using the blocks as its states. Let γ be the transition function of B. Suppose S is a set of equivalent states of A, and a is an input symbol. Then there must exist one block T of states such that for all states q in S, $\delta(q, a)$ is a member of block T. For if not, then input symbol a takes two states p and q of S to states in different blocks, and those states are distinguishable by Theorem 4.24. That fact lets us conclude that p and q are not equivalent, and they did not both belong in S. As a consequence, we can let $\gamma(S, a) = T$. In addition:
   (a) The start state of B is the block containing the start state of A.
   (b) The set of accepting states of B is the set of blocks containing accepting states of A. Note that if one state of a block is accepting, then all the states of that block must be accepting. The reason is that any accepting state is distinguishable from any nonaccepting state, so you can't have both accepting and nonaccepting states in one block of equivalent states.

Example 4.25: Let us minimize the DFA from Fig. 4.8. We established the blocks of the state partition in Example 4.22. Figure 4.12 shows the minimum-state automaton. Its five states correspond to the five blocks of equivalent states for the automaton of Fig. 4.8.

The start state is $\{A, E\}$, since A was the start state of Fig. 4.8. The only accepting state is $\{C\}$, since C is the only accepting state of Fig. 4.8. Notice that the transitions of Fig. 4.12 properly reflect the transitions of Fig. 4.8. For instance, Fig. 4.12 has a transition on input 0 from $\{A, E\}$ to $\{B, H\}$. That

[5] You should remember that the same block may be formed several times, starting from different states. However, the partition consists of the *different* blocks, so this block appears only once in the partition.

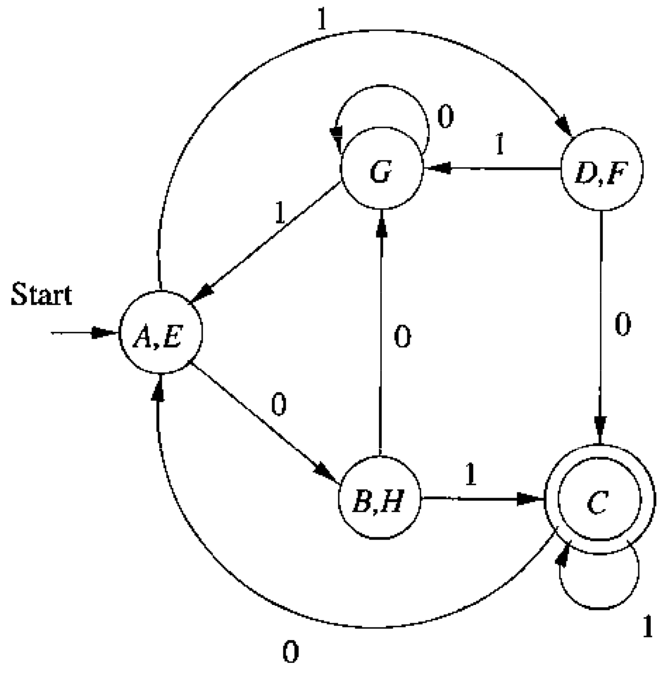


Figure 4.12: Minimum-state DFA equivalent to Fig. 4.8

makes sense, because in Fig. 4.8, A goes to B on input 0, and E goes to H. Likewise, on input 1, $\{A, E\}$ goes to $\{D, F\}$. If we examine Fig. 4.8, we find that both A and E go to F on input 1, so the selection of the successor of $\{A, E\}$ on input 1 is also correct. Note that the fact neither A nor E goes to D on input 1 is not important. You may check that all of the other transitions are also proper. □

4.4.4 Why the Minimized DFA Can't Be Beaten

Suppose we have a DFA A, and we minimize it to construct a DFA M, using the partitioning method of Theorem 4.24. That theorem shows that we can't group the states of A into fewer groups and still have an equivalent DFA. However, could there be another DFA N, unrelated to A, that accepts the same language as A and M, yet has fewer states than M? We can prove by contradiction that N does not exist.

First, run the state-distinguishability process of Section 4.4.1 on the states of M and N together, as if they were one DFA. We may assume that the states of M and N have no names in common, so the transition function of the combined automaton is the union of the transition rules of M and N, with no interaction. States are accepting in the combined DFA if and only if they are accepting in the DFA from which they come.

The start states of M and N are indistinguishable because $L(M) = L(N)$. Further, if $\{p, q\}$ are indistinguishable, then their successors on any one input

Minimizing the States of an NFA

You might imagine that the same state-partition technique that minimizes the states of a DFA could also be used to find a minimum-state NFA equivalent to a given NFA or DFA. While we can, by a process of exhaustive enumeration, find an NFA with as few states as possible accepting a given regular language, we cannot simply group the states of some given NFA for the language.

An example is in Fig. 4.13. None of the three states are equivalent. Surely accepting state B is distinguishable from nonaccepting states A and C. However, A and C are distinguishable by input 0. The successors of C are A alone, which does not include an accepting state, while the successors of A are $\{A, B\}$, which does include an accepting state. Thus, grouping equivalent states does not reduce the number of states of Fig. 4.13.

However, we can find a smaller NFA for the same language if we simply remove state C. Note that A and B alone accept all strings ending in 0, while adding state C does not allow us to accept any other strings.

symbol are also indistinguishable. The reason is that if we could distinguish the successors, then we could distinguish p from q.

Neither M nor N could have an inaccessible state, or else we could eliminate that state and have an even smaller DFA for the same language. Thus, every state of M is indistinguishable from at least one state of N. To see why, suppose p is a state of M. Then there is some string $a_1a_2\cdots a_k$ that takes the start state of M to state p. This string also takes the start state of N to some state q. Since we know the start states are indistinguishable, we also know that their successors under input symbol a_1 are indistinguishable. Then, the successors of those states on input a_2 are indistinguishable, and so on, until we conclude

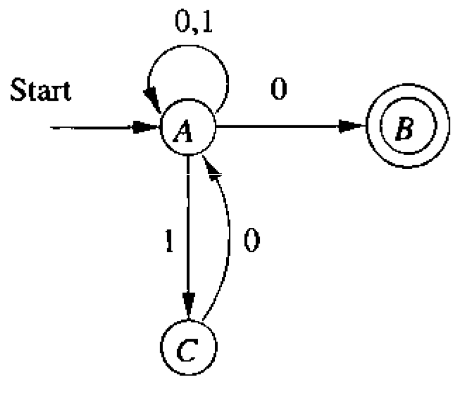


Figure 4.13: An NFA that cannot be minimized by state equivalence

that p and q are indistinguishable.

Since N has fewer states than M, there are two states of M that are indistinguishable from the same state of N, and therefore indistinguishable from each other. But M was designed so that all its states *are* distinguishable from each other. We have a contradiction, so the assumption that N exists is wrong, and M in fact has as few states as any equivalent DFA for A. Formally, we have proved:

Theorem 4.26: If A is a DFA, and M the DFA constructed from A by the algorithm described in the statement of Theorem 4.24, then M has as few states as any DFA equivalent to A. □

In fact we can say something even stronger than Theorem 4.26. There must be a one-to-one correspondence between the states of any other minimum-state N and the DFA M. The reason is that we argued above how each state of M must be equivalent to one state of N, and no state of M can be equivalent to two states of N. We can similarly argue that no state of N can be equivalent to two states of M, although each state of N must be equivalent to one of M's states. Thus, the minimum-state DFA equivalent to A is unique except for a possible renaming of the states.

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

Figure 4.14: A DFA to be minimized

4.4.5 Exercises for Section 4.4

* **Exercise 4.4.1:** In Fig. 4.14 is the transition table of a DFA.

a) Draw the table of distinguishabilities for this automaton.

b) Construct the minimum-state equivalent DFA.

Exercise 4.4.2: Repeat Exercise 4.4.1 for the DFA of Fig 4.15.

!! **Exercise 4.4.3:** Suppose that p are q are distinguishable states of a given DFA A with n states. As a function of n, what is the tightest upper bound on how long the shortest string that distinguishes p from q can be?