An Introduction to

FORMAL LANGUAGES and AUTOMATA

Fifth Edition

PETER LINZ

INCLUDES CD-ROM

مادة محمية بموجب حقوق النشر

World Headquarters
Jones & Bartlett Learning
40 Tall Pine Drive
Sudbury, MA 01776
978-443-5000
info@jblearning.com
www.jblearning.com

Jones & Bartlett Learning
Canada
6339 Ormindale Way
Mississauga, Ontario L5V 1J2
Canada

Jones & Bartlett Learning
International
Barb House, Barb Mews
London W6 7PA
United Kingdom

Jones & Bartlett Learning books and products are available through most bookstores and online booksellers. To contact Jones & Bartlett Learning directly, call 800-832-0034, fax 978-443-8000, or visit our website, www.jblearning.com.

Substantial discounts on bulk quantities of Jones & Bartlett Learning publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones & Bartlett Learning via the above contact information or send an email to specialsales@jblearning.com.

Copyright © 2012 by Jones & Bartlett Learning, LLC

All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

Production Credits
Publisher: Cathleen Sether
Senior Acquisitions Editor: Timothy Anderson
Senior Editorial Assistant: Stephanie Sguigna
Production Director: Amy Rose
Senior Marketing Manager: Andrea DeFronzo
V.P., Manufacturing and Inventory Control: Therese Connell
Composition: Northeast Compositors, Inc.
Cover and Title Page Design: Kristin E. Parker
Cover Image: © Alexis Puentes/ShutterStock, Inc.
Printing and Binding: Malloy, Inc.
Cover Printing: Malloy, Inc.

Library of Congress Cataloging-in-Publication Data

Linz, Peter.
An introduction to formal languages and automata / Peter Linz. — 5th ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4496-1552-9 (casebound)
1. Formal languages. 2. Machine theory. I. Title.
QA267.3.L56 2011
005.13'1—dc22

2010040050

6048

Printed in the United States of America
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

مادة محمية بموجب حقوق النشر

Contents

مادة محمية بموجب حقوق النشر

مادة محمية بموجب حقوق النشر

"15529˙TOCx˙Linz" — 2011/1/13 — 12:21 — page vii — #3

مادة محمية بموجب حقوق النشر

مادة محمية بموجب حقوق النشر

مادة محمية بموجب حقوق النشر

مادة محمية بموجب حقوق النشر

Preface

This book is designed for an introductory course on formal languages, automata, computability, and related matters. These topics form a major part of what is known as the theory of computation. A course on this subject matter is now standard in the computer science curriculum and is often taught fairly early in the program. Hence, the prospective audience for this book consists primarily of sophomores and juniors majoring in computer science or computer engineering.

Prerequisites for the material in this book are a knowledge of some higher-level programming language (commonly C, C++, or Java™) and familiarity with the fundamentals of data structures and algorithms. A course in discrete mathematics that includes set theory, functions, relations, logic, and elements of mathematical reasoning is essential. Such a course is part of the standard introductory computer science curriculum.

The study of the theory of computation has several purposes, most importantly (1) to familiarize students with the foundations and principles of computer science, (2) to teach material that is useful in subsequent courses, and (3) to strengthen students' ability to carry out formal and rigorous

مادة محمية بموجب حقوق النشر

mathematical arguments. The presentation I have chosen for this text favors the first two purposes, although I would argue that it also serves the third. To present ideas clearly and to give students insight into the material, the text stresses intuitive motivation and illustration of ideas through examples. When there is a choice, I prefer arguments that are easily grasped to those that are concise and elegant but difficult in concept. I state definitions and theorems precisely and give the motivation for proofs, but often leave out the routine and tedious details. I believe that this is desirable for pedagogical reasons. Many proofs are unexciting applications of induction or contradiction with differences that are specific to particular problems. Presenting such arguments in full detail is not only unnecessary, but interferes with the flow of the story. Therefore, quite a few of the proofs are brief and someone who insists on completeness may consider them lacking in detail. I do not see this as a drawback. Mathematical skills are not the byproduct of reading someone else's arguments, but come from thinking about the essence of a problem, discovering ideas suitable to make the point, then carrying them out in precise detail. The latter skill certainly has to be learned, and I think that the proof sketches in this text provide very appropriate starting points for such a practice.

Computer science students sometimes view a course in the theory of computation as unnecessarily abstract and of no practical consequence. To convince them otherwise, one needs to appeal to their specific interests and strengths, such as tenacity and inventiveness in dealing with hard-to-solve problems. Because of this, my approach emphasizes learning through problem solving.

By a problem-solving approach, I mean that students learn the material primarily through problem-type illustrative examples that show the motivation behind the concepts, as well as their connection to the theorems and definitions. At the same time, the examples may involve a nontrivial aspect, for which students must discover a solution. In such an approach, homework exercises contribute to a major part of the learning process. The exercises at the end of each section are designed to illuminate and illustrate the material and call on students' problem-solving ability at various levels. Some of the exercises are fairly simple, picking up where the discussion in the text leaves off and asking students to carry on for another step or two. Other exercises are very difficult, challenging even the best minds. The more difficult exercises are marked with a star. A good mix of such exercises can be a very effective teaching tool. Students need not be asked to solve all problems, but should be assigned those that support the goals of the course and the viewpoint of the instructor. Computer science curricula differ from institution to institution; while a few emphasize the theoretical side, others are almost entirely oriented toward practical application. I believe that this text can serve either of these extremes, provided that the exercises are selected carefully with the students' background and interests in mind. At the same time, the instructor needs to inform the students about the level of

مادة محمية بموجب حقوق النشر

abstraction that is expected of them. This is particularly true of the proof-oriented exercises. When I say "prove that" or "show that," I have in mind that the student should think about how a proof can be constructed and then produce a clear argument. How formal such a proof should be needs to be determined by the instructor, and students should be given guidelines on this early in the course.

The content of the text is appropriate for a one-semester course. Most of the material can be covered, although some choice of emphasis will have to be made. In my classes, I generally gloss over proofs, giving just enough coverage to make the result plausible, and then ask students to read the rest on their own. Overall, though, little can be skipped entirely without potential difficulties later on. A few sections, which are marked with an asterisk, can be omitted without loss to later material. Most of the material, however, is essential and must be covered.

The fifth edition of this text introduces a substantial amount of new material. While the presentation in the fourth edition has been retained with only minor modifications, two appendices have been added. The first is an entire chapter on finite-state transducers, Appendix A. While transducers play no significant role in formal language theory, they are important in other areas of computer science, such as digital design. Students can benefit from an early exposure to this subject; if time permits it is worthwhile to do so. Due to the similarity with finite accepters, this involves few new concepts.

I also added an introduction to JFLAP, an interactive software tool that I feel is of great help in both learning the material and in teaching this course. JFLAP implements most of the ideas and constructions in this book. It not only helps students visualize abstract concepts, but it is also a great time-saver. Many of the exercises in this book require creating structures that are complicated and that have to be thoroughly tested for correctness. JFLAP can reduce the time required for this by an order of magnitude. Appendix B gives a brief introduction to JFLAP and the CD that comes with the book expands on this. I very much recommend the use of JFLAP for both students and instructors.

Peter Linz

مادة محمية بموجب حقوق النشر

مادة محمية بموجب حقوق النشر

Chapter 1

Introduction to the Theory of Computation

The subject matter of this book, the theory of computation, includes several topics: automata theory, formal languages and grammars, computability, and complexity. Together, this material constitutes the theoretical foundation of computer science. Loosely speaking we can think of automata, grammars, and computability as the study of what can be done by computers in principle, while complexity addresses what can be done in practice. In this book we focus almost entirely on the first of these concerns. We will study various automata, see how they are related to languages and grammars, and investigate what can and cannot be done by digital computers. Although this theory has many uses, it is inherently abstract and mathematical.

Computer science is a practical discipline. Those who work in it often have a marked preference for useful and tangible problems over theoretical speculation. This is certainly true of computer science students who are concerned mainly with difficult applications from the real world. Theoretical questions interest them only if they help in finding good solutions. This attitude is appropriate, since without applications there would be little interest in computers. But given this practical orientation, one might well ask "why study theory?"

مادة محمية بموجب حقوق النشر

The first answer is that theory provides concepts and principles that help us understand the general nature of the discipline. The field of computer science includes a wide range of special topics, from machine design to programming. The use of computers in the real world involves a wealth of specific detail that must be learned for a successful application. This makes computer science a very diverse and broad discipline. But in spite of this diversity, there are some common underlying principles. To study these basic principles, we construct abstract models of computers and computation. These models embody the important features that are common to both hardware and software and that are essential to many of the special and complex constructs we encounter while working with computers. Even when such models are too simple to be applicable immediately to real-world situations, the insights we gain from studying them provide the foundation on which specific development is based. This approach is, of course, not unique to computer science. The construction of models is one of the essentials of any scientific discipline, and the usefulness of a discipline is often dependent on the existence of simple, yet powerful, theories and laws.

A second, and perhaps not so obvious, answer is that the ideas we will discuss have some immediate and important applications. The fields of digital design, programming languages, and compilers are the most obvious examples, but there are many others. The concepts we study here run like a thread through much of computer science, from operating systems to pattern recognition.

The third answer is one of which we hope to convince the reader. The subject matter is intellectually stimulating and fun. It provides many challenging, puzzle-like problems that can lead to some sleepless nights. This is problem solving in its pure essence.

In this book, we will look at models that represent features at the core of all computers and their applications. To model the hardware of a computer, we introduce the notion of an **automaton** (plural, **automata**). An automaton is a construct that possesses all the indispensable features of a digital computer. It accepts input, produces output, may have some temporary storage, and can make decisions in transforming the input into the output. A **formal language** is an abstraction of the general characteristics of programming languages. A formal language consists of a set of symbols and some rules of formation by which these symbols can be combined into entities called sentences. A formal language is the set of all sentences permitted by the rules of formation. Although some of the formal languages we study here are simpler than programming languages, they have many of the same essential features. We can learn a great deal about programming languages from formal languages. Finally, we will formalize the concept of a mechanical computation by giving a precise definition of the term **algorithm** and study the kinds of problems that are (and are not) suitable for solution by such mechanical means. In the course of our study, we will

مادة محمية بموجب حقوق النشر

show the close connection between these abstractions and investigate the conclusions we can derive from them.

In the first chapter, we look at these basic ideas in a very broad way to set the stage for later work. In Section 1.1, we review the main ideas from mathematics that will be required. While intuition will frequently be our guide in exploring ideas, the conclusions we draw will be based on rigorous arguments. This will involve some mathematical machinery, although the requirements are not extensive. The reader will need a reasonably good grasp of the terminology and of the elementary results of set theory, functions, and relations. Trees and graph structures will be used frequently, although little is needed beyond the definition of a labeled, directed graph. Perhaps the most stringent requirement is the ability to follow proofs and an understanding of what constitutes proper mathematical reasoning. This includes familiarity with the basic proof techniques of deduction, induction, and proof by contradiction. We will assume that the reader has this necessary background. Section 1.1 is included to review some of the main results that will be used and to establish a notational common ground for subsequent discussion.

In Section 1.2, we take a first look at the central concepts of languages, grammars, and automata. These concepts occur in many specific forms throughout the book. In Section 1.3, we give some simple applications of these general ideas to illustrate that these concepts have widespread uses in computer science. The discussion in these two sections will be intuitive rather than rigorous. Later, we will make all of this much more precise; but for the moment, the goal is to get a clear picture of the concepts with which we are dealing.

1.1 Mathematical Preliminaries and Notation

Sets

A **set** is a collection of elements, without any structure other than membership. To indicate that x is an element of the set S, we write $x \in S$. The statement that x is not in S is written $x \notin S$. A set can be specified by enclosing some description of its elements in curly braces; for example, the set of integers 0, 1, 2 is shown as

$$S = \{0, 1, 2\}.$$

Ellipses are used whenever the meaning is clear. Thus, $\{a, b, \ldots, z\}$ stands for all the lowercase letters of the English alphabet, while $\{2, 4, 6, \ldots\}$ denotes the set of all positive even integers. When the need arises, we use more explicit notation, in which we write

$$S = \{i : i > 0, i \text{ is even}\} \tag{1.1}$$

مادة محمية بموجب حقوق النشر

for the last example. We read this as "S is the set of all i, such that i is greater than zero, and i is even," implying, of course, that i is an integer.

The usual set operations are **union** ($\cup$), **intersection** ($\cap$), and **difference** ($-$) defined as

$$\begin{aligned} S_1 \cup S_2 &= \{x : x \in S_1 \text{ or } x \in S_2\}, \\ S_1 \cap S_2 &= \{x : x \in S_1 \text{ and } x \in S_2\}, \\ S_1 - S_2 &= \{x : x \in S_1 \text{ and } x \notin S_2\}. \end{aligned}$$

Another basic operation is **complementation**. The complement of a set S, denoted by $\overline{S}$, consists of all elements not in S. To make this meaningful, we need to know what the **universal set** U of all possible elements is. If U is specified, then

$$\overline{S} = \{x : x \in U, x \notin S\}.$$

The set with no elements, called the **empty set** or the **null set**, is denoted by $\varnothing$. From the definition of a set, it is obvious that

$$\begin{aligned} S \cup \varnothing &= S - \varnothing = S, \\ S \cap \varnothing &= \varnothing, \\ \overline{\varnothing} &= U, \\ \overline{\overline{S}} &= S. \end{aligned}$$

The following useful identities, known as **DeMorgan's laws**,

$$\overline{S_1 \cup S_2} = \overline{S}_1 \cap \overline{S}_2, \tag{1.2}$$

$$\overline{S_1 \cap S_2} = \overline{S}_1 \cup \overline{S}_2, \tag{1.3}$$

are needed on several occasions.

A set S_1 is said to be a **subset** of S if every element of S_1 is also an element of S. We write this as

$$S_1 \subseteq S.$$

If $S_1 \subseteq S$, but S contains an element not in S_1, we say that S_1 is a **proper subset** of S; we write this as

$$S_1 \subset S.$$

If S_1 and S_2 have no common element, that is, $S_1 \cap S_2 = \varnothing$, then the sets are said to be **disjoint**.

A set is said to be **finite** if it contains a finite number of elements; otherwise it is **infinite**. The size of a finite set is the number of elements in it; this is denoted by $|S|$.

مادة محمية بموجب حقوق النشر

A given set normally has many subsets. The set of all subsets of a set S is called the **powerset** of S and is denoted by 2^S. Observe that 2^S is a set of sets.

Example 1.1 If S is the set $\{a, b, c\}$, then its powerset is

$$2^S = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Here $|S| = 3$ and $\left|2^S\right| = 8$. This is an instance of a general result; if S is finite, then

$$\left|2^S\right| = 2^{|S|}.$$

In many of our examples, the elements of a set are ordered sequences of elements from other sets. Such sets are said to be the **Cartesian product** of other sets. For the Cartesian product of two sets, which itself is a set of ordered pairs, we write

$$S = S_1 \times S_2 = \{(x, y) : x \in S_1, y \in S_2\}.$$

Example 1.2 Let $S_1 = \{2, 4\}$ and $S_2 = \{2, 3, 5, 6\}$. Then

$$S_1 \times S_2 = \{(2, 2), (2, 3), (2, 5), (2, 6), (4, 2), (4, 3), (4, 5), (4, 6)\}.$$

Note that the order in which the elements of a pair are written matters. The pair $(4, 2)$ is in $S_1 \times S_2$, but $(2, 4)$ is not.

The notation is extended in an obvious fashion to the Cartesian product of more than two sets; generally

$$S_1 \times S_2 \times \cdots \times S_n = \{(x_1, x_2, ..., x_n) : x_i \in S_i\}.$$

A set can be divided by separating it into a number of subsets. Suppose that S_1, S_2, ..., S_n are subsets of a given set S and that the following holds:

1. The subsets S_1, S_2, ..., S_n are mutually disjoint;
2. $S_1 \cup S_2 \cup ... \cup S_n = S$;
3. none of the S_i is empty.

Then S_1, S_2, ..., S_n is called a **partition** of S.

مادة محمية بموجب حقوق النشر

Functions and Relations

A **function** is a rule that assigns to elements of one set a unique element of another set. If f denotes a function, then the first set is called the **domain** of f, and the second set is its **range**. We write

$$f : S_1 \to S_2$$

to indicate that the domain of f is a subset of S_1 and that the range of f is a subset of S_2. If the domain of f is all of S_1, we say that f is a **total function** on S_1; otherwise f is said to be a **partial function**.

In many applications, the domain and range of the functions involved are in the set of positive integers. Furthermore, we are often interested only in the behavior of these functions as their arguments become very large. In such cases an understanding of the growth rates may suffice and a common order of magnitude notation can be used. Let $f(n)$ and $g(n)$ be functions whose domain is a subset of the positive integers. If there exists a positive constant c such that for all sufficiently large n

$$f(n) \leq c|g(n)|,$$

we say that f has **order at most** g. We write this as

$$f(n) = O(g(n)).$$

If

$$|f(n)| \geq c\,|g(n)|,$$

then f has **order at least** g, for which we use

$$f(n) = \Omega(g(n)).$$

Finally, if there exist constants c_1 and c_2 such that

$$c_1\,|g(n)| \leq |f(n)| \leq c_2\,|g(n)|,$$

f and g have the **same order of magnitude**, expressed as

$$f(n) = \Theta(g(n)).$$

In this order-of-magnitude notation, we ignore multiplicative constants and lower-order terms that become negligible as n increases.

Example 1.3 Let

$$\begin{aligned} f(n) &= 2n^2 + 3n, \\ g(n) &= n^3, \\ h(n) &= 10n^2 + 100. \end{aligned}$$

مادة محمية بموجب حقوق النشر

Then

$$\begin{aligned} f(n) &= O(g(n)), \\ g(n) &= \Omega(h(n)), \\ f(n) &= \Theta(h(n)). \end{aligned}$$

In order-of-magnitude notation, the symbol = should not be interpreted as equality and order-of-magnitude expressions cannot be treated like ordinary expressions. Manipulations such as

$$O(n) + O(n) = 2O(n)$$

are not sensible and can lead to incorrect conclusions. Still, if used properly, the order-of-magnitude arguments can be effective, as we will see in later chapters.

■

Some functions can be represented by a set of pairs

$$\{(x_1, y_1), (x_2, y_2), ...\},$$

where x_i is an element in the domain of the function, and y_i is the corresponding value in its range. For such a set to define a function, each x_i can occur at most once as the first element of a pair. If this is not satisfied, the set is called a **relation**. Relations are more general than functions: In a function each element of the domain has exactly one associated element in the range; in a relation there may be several such elements in the range.

One kind of relation is that of **equivalence**, a generalization of the concept of equality (identity). To indicate that a pair (x, y) is in an equivalence relation, we write

$$x \equiv y.$$

A relation denoted by $\equiv$ is considered an equivalence if it satisfies three rules: the reflexivity rule

$$x \equiv x \text{ for all } x;$$

the symmetry rule

$$\text{if } x \equiv y, \text{ then } y \equiv x;$$

and the transitivity rule

$$\text{if } x \equiv y \text{ and } y \equiv z, \text{ then } x \equiv z.$$

مادة محمية بموجب حقوق النشر

Example 1.4 On the set of nonnegative integers, we can define a relation

$$x \equiv y$$

if and only if

$$x \bmod 3 = y \bmod 3.$$

Then $2 \equiv 5$, $12 \equiv 0$, and $0 \equiv 36$. Clearly this is an equivalence relation, as it satisfies reflexivity, symmetry, and transitivity. ■

If S is a set on which we have a defined equivalence relation, then we can use this equivalence to partition the set into **equivalence classes**. Each equivalence class contains all and only equivalent elements.

Graphs and Trees

A graph is a construct consisting of two finite sets, the set $V = \{v_1, v_2, ..., v_n\}$ of **vertices** and the set $E = \{e_1, e_2, ..., e_m\}$ of **edges**. Each edge is a pair of vertices from V, for instance,

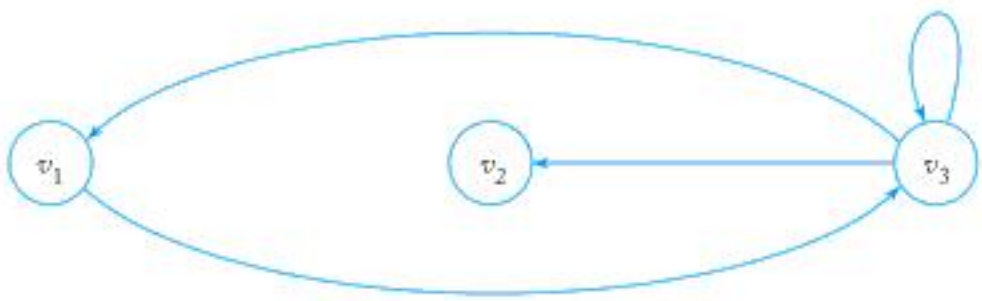
$$e_i = (v_j, v_k)$$

is an edge from v_j to v_k. We say that the edge e_i is an outgoing edge for v_j and an incoming edge for v_k. Such a construct is actually a directed graph (digraph), since we associate a direction (from v_j to v_k) with each edge. Graphs may be labeled, a label being a name or other information associated with parts of the graph. Both vertices and edges may be labeled.

Graphs are conveniently visualized by diagrams in which the vertices are represented as circles and the edges as lines with arrows connecting the vertices. The graph with vertices $\{v_1, v_2, v_3\}$ and edges $\{(v_1, v_3), (v_3, v_1), (v_3, v_2), (v_3, v_3)\}$ is depicted in Figure 1.1.

A sequence of edges $(v_i, v_j), (v_j, v_k), \ldots, (v_m, v_n)$ is said to be a **walk** from v_i to v_n. The length of a walk is the total number of edges traversed in going from the initial vertex to the final one. A walk in which no edge is repeated is said to be a **path**; a path is **simple** if no vertex is repeated. A walk from v_i to itself with no repeated edges is called a **cycle** with **base**

Figure 1.1



مادة محمية بموجب حقوق النشر

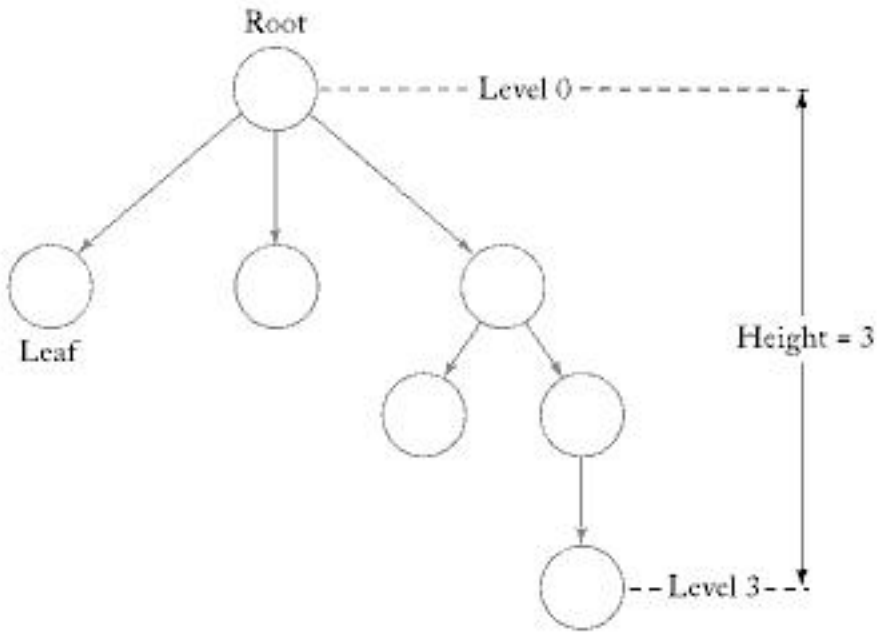
v_i. If no vertices other than the base are repeated in a cycle, then it is said to be simple. In Figure 1.1, (v_1, v_3), (v_3, v_2) is a simple path from v_1 to v_2. The sequence of edges (v_1, v_3), (v_3, v_3), (v_3, v_1) is a cycle, but not a simple one. If the edges of a graph are labeled, we can talk about the label of a walk. This label is the sequence of edge labels encountered when the path is traversed. Finally, an edge from a vertex to itself is called a **loop**. In Figure 1.1, there is a loop on vertex v_3.

On several occasions, we will refer to an algorithm for finding all simple paths between two given vertices (or all simple cycles based on a vertex). If we do not concern ourselves with efficiency, we can use the following obvious method. Starting from the given vertex, say v_i, list all outgoing edges (v_i, v_k), (v_i, v_l), ... At this point, we have all paths of length one starting at v_i. For all vertices $v_k, v_l, \ldots$ so reached, we list all outgoing edges as long as they do not lead to any vertex already used in the path we are constructing. After we do this, we will have all simple paths of length two originating at v_i. We continue this until all possibilities are accounted for. Since there are only a finite number of vertices, we will eventually list all simple paths beginning at v_i. From these we select those ending at the desired vertex.

Trees are a particular type of graph. A tree is a directed graph that has no cycles, and that has one distinct vertex, called the **root**, such that there is exactly one path from the root to every other vertex. This definition implies that the root has no incoming edges and that there are some vertices without outgoing edges. These are called the **leaves** of the tree. If there is an edge from v_i to v_j, then v_i is said to be the **parent** of v_j, and v_j the **child** of v_i. The **level** associated with each vertex is the number of edges in the path from the root to the vertex. The **height** of the tree is the largest level number of any vertex. These terms are illustrated in Figure 1.2.

At times, we want to associate an ordering with the nodes at each level; in such cases we talk about **ordered trees**.

Figure 1.2



مادة محمية بموجب حقوق النشر

More details on graphs and trees can be found in most books on discrete mathematics.

Proof Techniques

An important requirement for reading this text is the ability to follow proofs. In mathematical arguments, we employ the accepted rules of deductive reasoning, and many proofs are simply a sequence of such steps. Two special proof techniques are used so frequently that it is appropriate to review them briefly. These are **proof by induction** and **proof by contradiction**.

Induction is a technique by which the truth of a number of statements can be inferred from the truth of a few specific instances. Suppose we have a sequence of statements $P_1, P_2, ...$ we want to prove to be true. Furthermore, suppose also that the following holds:

1. For some $k \geq 1$, we know that $P_1, P_2, ... , P_k$ are true.

2. The problem is such that for any $n \geq k$, the truths of $P_1, P_2, ..., P_n$ imply the truth of P_{n+1}.

We can then use induction to show that every statement in this sequence is true.

In a proof by induction, we argue as follows: From Condition 1 we know that the first k statements are true. Then Condition 2 tells us that P_{k+1} also must be true. But now that we know that the first $k+1$ statements are true, we can apply Condition 2 again to claim that P_{k+2} must be true, and so on. We need not explicitly continue this argument, because the pattern is clear. The chain of reasoning can be extended to any statement. Therefore, every statement is true.

The starting statements $P_1, P_2, ... P_k$ are called the **basis** of the induction. The step connecting P_n with P_{n+1} is called the **inductive step**. The inductive step is generally made easier by the **inductive assumption** that $P_1, P_2, ..., P_n$ are true, then argue that the truth of these statements guarantees the truth of P_{n+1}. In a formal inductive argument, we show all three parts explicitly.

Example 1.5 A binary tree is a tree in which no parent can have more than two children. Prove that a binary tree of height n has at most 2^n leaves.

Proof: If we denote the maximum number of leaves of a binary tree of height n by $l(n)$, then we want to show that $l(n) \leq 2^n$.

Basis: Clearly $l(0) = 1 = 2^0$ since a tree of height 0 can have no nodes other than the root, that is, it has at most one leaf.

مادة محمية بموجب حقوق النشر

Inductive Assumption:

$$l(i) \leq 2^i, \text{for } i = 0, 1, ..., n. \tag{1.4}$$

Inductive Step: To get a binary tree of height $n+1$ from one of height n, we can create, at most, two leaves in place of each previous one. Therefore,

$$l(n+1) = 2l(n).$$

Now, using the inductive assumption, we get

$$l(n+1) \leq 2 \times 2^n = 2^{n+1}.$$

Thus, if our claim is true for n, it must also be true for $n+1$. Since n can be any number, the statement must be true for all n. ■

Here we introduce the symbol ■ that is used in this book to denote the end of a proof.

Inductive reasoning can be difficult to grasp. It helps to notice the close connection between induction and recursion in programming. For example, the recursive definition of a function $f(n)$, where n is any positive integer, often has two parts. One involves the definition of $f(n+1)$ in terms of $f(n), f(n-1), ..., f(1)$. This corresponds to the inductive step. The second part is the "escape" from the recursion, which is accomplished by defining $f(1), f(2), ..., f(k)$ nonrecursively. This corresponds to the basis of induction. As in induction, recursion allows us to draw conclusions about all instances of the problem, given only a few starting values and using the recursive nature of the problem.

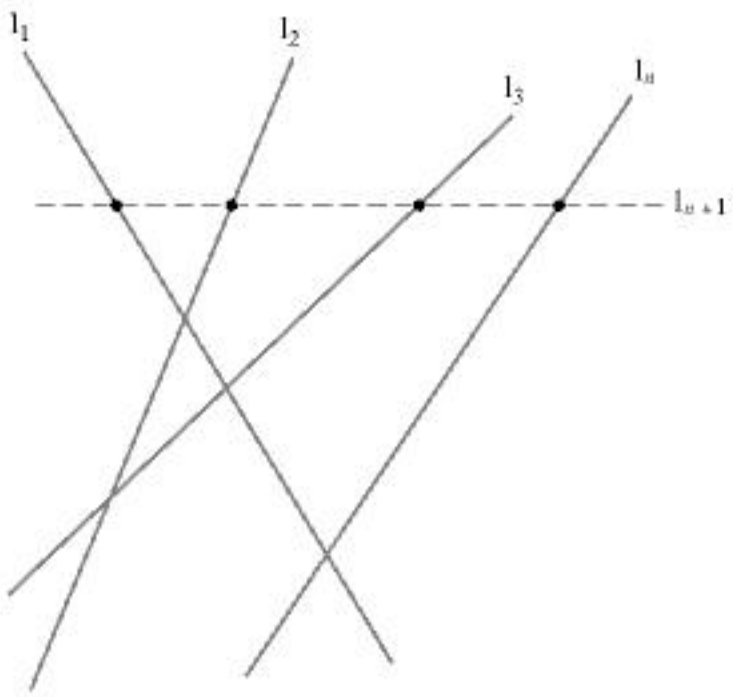
Sometimes, a problem looks difficult until we look at it in just the right way. Often looking at it recursively simplifies matters greatly.

Example 1.6 A set $l_1, l_2, ..., l_n$ of mutually intersecting straight lines divides the plane into a number of separated regions. A single line divides the plane into two parts, two lines generate four regions, three lines make seven regions, and so on. This is easily checked visually for up to three lines, but as the number of lines increases it becomes difficult to spot a pattern. Let us try to solve this problem recursively.

Look at Figure 1.3 to see what happens if we add a new line l_{n+1} to existing n lines. The region to the left of l_1 is divided into two new regions, so is the region to the left of l_2, and so on until we get to the last line. At the last line, the region to the right of l_n is also divided. Each of the n intersections then generates one new region, with one extra at the end. So,

مادة محمية بموجب حقوق النشر

Figure 1.3



if we let $A(n)$ denote the number of regions generated by n lines, we see that

$$A(n+1) = A(n) + n + 1, n = 1, 2, ...,$$

with $A(1) = 2$. From this simple recursion we then calculate $A(2) = 4$, $A(3) = 7$, $A(4) = 11$, and so on.

To get a formula for $A(n)$ and to show that it is correct, we use induction. If we conjecture that

$$A(n) = \frac{n(n+1)}{2} + 1,$$

then

$$\begin{aligned} A(n+1) &= \frac{n(n+1)}{2} + 1 + n + 1 \\ &= \frac{(n+1)(n+2)}{2} + 1 \end{aligned}$$

justifies the inductive step. The basis is easily checked, completing the argument.

In this example we have been a little less formal in identifying the basis, inductive assumption, and inductive step, but they are there and are essential. To keep our subsequent discussions from becoming too formal, we will generally prefer the style of this second example. However, if you have difficulty in following or constructing a proof, go back to the more explicit form of Example 1.5.

Proof by contradiction is another powerful technique that often works when everything else fails. Suppose we want to prove that some statement

مادة محمية بموجب حقوق النشر

P is true. We then assume, for the moment, that P is false and see where that assumption leads us. If we arrive at a conclusion that we know is incorrect, we can lay the blame on the starting assumption and conclude that P must be true. The following is a classic and elegant example.

Example 1.7 A rational number is a number that can be expressed as the ratio of two integers n and m so that n and m have no common factor. A real number that is not rational is said to be irrational. Show that $\sqrt{2}$ is irrational.

As in all proofs by contradiction, we assume the contrary of what we want to show. Here we assume that $\sqrt{2}$ is a rational number so that it can be written as

$$\sqrt{2} = \frac{n}{m}, \tag{1.5}$$

where n and m are integers without a common factor. Rearranging (1.5), we have

$$2m^2 = n^2.$$

Therefore, n^2 must be even. This implies that n is even, so that we can write $n = 2k$ or

$$2m^2 = 4k^2,$$

and

$$m^2 = 2k^2.$$

Therefore, m is even. But this contradicts our assumption that n and m have no common factors. Thus, m and n in (1.5) cannot exist and $\sqrt{2}$ is not a rational number.

This example exhibits the essence of a proof by contradiction. By making a certain assumption we are led to a contradiction of the assumption or some known fact. If all steps in our argument are logically sound, we must conclude that our initial assumption was false.

EXERCISES

1. Use induction on the size of S to show that if S is a finite set, then $\left|2^S\right| = 2^{|S|}$.
2. Show that if S_1 and S_2 are finite sets with $|S_1| = n$ and $|S_2| = m$, then
$$|S_1 \cup S_2| \leq n + m.$$
3. If S_1 and S_2 are finite sets, show that $|S_1 \times S_2| = |S_1||S_2|$.
4. Consider the relation between two sets defined by $S_1 \equiv S_2$ if and only if $|S_1| = |S_2|$. Show that this is an equivalence relation.

مادة محمية بموجب حقوق النشر

5. Prove DeMorgan's laws, Equations (1.2) and (1.3).
6. Occasionally, we need to use the union and intersection symbols in a manner analogous to the summation sign Σ. We define

$$\bigcup_{p\in\{i,j,k,\ldots\}} S_p = S_i \cup S_j \cup S_k \cdots$$

with an analogous notation for the intersection of several sets.
With this notation, the general DeMorgan's laws are written as

$$\overline{\bigcup_{p\in P} S_p} = \bigcap_{p\in P} \overline{S_p}$$

and

$$\overline{\bigcap_{p\in P} S_p} = \bigcup_{p\in P} \overline{S_p}.$$

Prove these identities when P is a finite set.
7. Show that

$$S_1 \cup S_2 = \overline{\overline{S}_1 \cap \overline{S}_2}.$$

8. Show that $S_1 = S_2$ if and only if

$$(S_1 \cap \overline{S}_2) \cup (\overline{S}_1 \cap S_2) = \varnothing.$$

9. Show that

$$S_1 \cup S_2 - (S_1 \cap \overline{S}_2) = S_2.$$

10. Show that the distributive law

$$S_1 \cap (S_2 \cup S_3) = (S_1 \cap S_2) \cup (S_1 \cap S_3)$$

holds for sets.
11. Show that

$$S_1 \times (S_2 \cup S_3) = (S_1 \times S_2) \cup (S_1 \times S_3).$$

12. Show that if $S_1 \subseteq S_2$, then $\overline{S}_2 \subseteq \overline{S}_1$.
13. Give conditions on S_1 and S_2 necessary and sufficient to ensure that

$$S_1 = (S_1 \cup S_2) - S_2.$$

14. Use the equivalence defined in Example 1.4 to partition the set $\{2, 4, 5, 6, 9, 23, 24, 25, 31, 37\}$ into equivalence classes.
15. Show that if $f(n) = O(g(n))$ and $g(n) = O(f(n))$, then $f(n) = \Theta(g(n))$.
16. Show that $2^n = O(3^n)$ but $2^n \neq \Theta(3^n)$.
17. Show that the following order-of-magnitude results hold.
    (a) $n^2 + 5\log n = O(n^2)$.
    (b) $3^n = O(n!)$.
    (c) $n! = O(n^n)$.

مادة محمية بموجب حقوق النشر

18. Prove that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

19. Show that if $f(n) = O(n^2)$ and $g(n) = O(n^3)$, then

$$f(n) + g(n) = O(n^3)$$

and

$$f(n)\, g(n) = O(n^6).$$

In this case, is it true that $g(n)/f(n) = O(n)$?

20. Assume that $f(n) = 2n^2 + n$ and $g(n) = O(n^2)$. What is wrong with the following argument?

$$f(n) = O(n^2) + O(n),$$

so that

$$f(n) - g(n) = O(n^2) + O(n) - O(n^2).$$

Therefore,

$$f(n) - g(n) = O(n).$$

21. Show that if $f(n) = \Theta(\log_2 n)$, then $f(n) = \Theta(\log_{10} n)$.

22. Draw a picture of the graph with vertices $\{v_1, v_2, v_3\}$ and edges $\{(v_1, v_1), (v_1, v_2), (v_2, v_3), (v_2, v_1), (v_3, v_1)\}$. Enumerate all cycles with base v_1.

23. Let $G = (V, E)$ be any graph. Prove the following claim: If there is any walk between $v_i \in V$ and $v_j \in V$, then there must be a path of length no larger than $|V| - 1$ between these two vertices.

24. Consider graphs in which there is at most one edge between any two vertices. Show that under this condition a graph with n vertices has at most n^2 edges.

25. Show that

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

26. Show that

$$\sum_{i=1}^{n} \frac{1}{i^2} \leq 2 - \frac{1}{n}.$$

27. Prove that for all $n \geq 4$ the inequality $2^n < n!$ holds.

28. The *Fibonacci sequence* is defined recursively by

$$f(n+2) = f(n+1) + f(n), n = 1, 2, \ldots,$$

with $f(1) = 1, f(2) = 1$. Show that
(a) $f(n) = O(2^n)$,
(b) $f(n) = \Omega(1.5^n)$.

29. Show that $\sqrt{8}$ is not a rational number.

30. Show that $2 - \sqrt{2}$ is irrational.

31. Show that $\sqrt{3}$ is irrational.

مادة محمية بموجب حقوق النشر

32. Prove or disprove the following statements.
    (a) The sum of a rational and an irrational number must be irrational.
    (b) The sum of two positive irrational numbers must be irrational.
    (c) The product of a non-zero rational and an irrational number must be irrational.
33. Show that every positive integer can be expressed as the product of prime numbers.
34. Prove that the set of all prime numbers is infinite.
35. A prime pair consists of two primes that differ by two. There are many prime pairs, for example, 11 and 13, 17 and 19, etc. Prime triplets are three numbers $n \geq 2, n+2, n+4$ that are all prime. Show that the only prime triplet is $(3, 5, 7)$.

1.2 Three Basic Concepts

Three fundamental ideas are the major themes of this book: **languages**, **grammars**, and **automata**. In the course of our study we will explore many results about these concepts and about their relationship to each other. First, we must understand the meaning of the terms.

Languages

We are all familiar with the notion of natural languages, such as English and French. Still, most of us would probably find it difficult to say exactly what the word "language" means. Dictionaries define the term informally as a system suitable for the expression of certain ideas, facts, or concepts, including a set of symbols and rules for their manipulation. While this gives us an intuitive idea of what a language is, it is not sufficient as a definition for the study of formal languages. We need a precise definition for the term.

We start with a finite, nonempty set Σ of symbols, called the **alphabet**. From the individual symbols we construct **strings**, which are finite sequences of symbols from the alphabet. For example, if the alphabet $\Sigma = \{a, b\}$, then $abab$ and $aaabbba$ are strings on Σ. With few exceptions, we will use lowercase letters $a, b, c, \ldots$ for elements of Σ and $u, v, w, \ldots$ for string names. We will write, for example,

$$w = abaaa$$

to indicate that the string named w has the specific value $abaaa$.

The **concatenation** of two strings w and v is the string obtained by appending the symbols of v to the right end of w, that is, if

$$w = a_1 a_2 \cdots a_n$$

and

$$v = b_1 b_2 \cdots b_m,$$

مادة محمية بموجب حقوق النشر

then the concatenation of w and v, denoted by wv, is

$$wv = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m.$$

The **reverse** of a string is obtained by writing the symbols in reverse order; if w is a string as shown above, then its reverse w^R is

$$w^R = a_n \cdots a_2 a_1.$$

The **length** of a string w, denoted by $|w|$, is the number of symbols in the string. We will frequently need to refer to the **empty string**, which is a string with no symbols at all. It will be denoted by λ. The following simple relations

$$\begin{aligned} |\lambda| &= 0, \\ \lambda w &= w\lambda = w \end{aligned}$$

hold for all w.

Any string of consecutive symbols in some w is said to be a **substring** of w. If

$$w = vu,$$

then the substrings v and u are said to be a **prefix** and a **suffix** of w, respectively. For example, if $w = abbab$, then $\{\lambda, a, ab, abb, abba, abbab\}$ is the set of all prefixes of w, while bab, ab, b are some of its suffixes.

Simple properties of strings, such as their length, are very intuitive and probably need little elaboration. For example, if u and v are strings, then the length of their concatenation is the sum of the individual lengths, that is,

$$|uv| = |u| + |v|. \tag{1.6}$$

But although this relationship is obvious, it is useful to be able to make it precise and prove it. The techniques for doing so are important in more complicated situations.

Example 1.8 Show that (1.6) holds for any u and v. To prove this, we first need a definition of the length of a string. We make such a definition in a recursive fashion by

$$\begin{aligned} |a| &= 1, \\ |wa| &= |w| + 1, \end{aligned}$$

for all $a \in \Sigma$ and w any string on Σ. This definition is a formal statement of our intuitive understanding of the length of a string: The length of a single

مادة محمية بموجب حقوق النشر

Example 1.10 If

$$L = \{a^n b^n : n \geq 0\},$$

then

$$L^2 = \{a^n b^n a^m b^m : n \geq 0, m \geq 0\}.$$

Note that n and m in the above are unrelated; the string $aabbaaabbb$ is in L^2.

The reverse of L is easily described in set notation as

$$L^R = \{b^n a^n : n \geq 0\},$$

but it is considerably harder to describe $\overline{L}$ or L^* this way. A few tries will quickly convince you of the limitation of set notation for the specification of complicated languages. ■

Grammars

To study languages mathematically, we need a mechanism to describe them. Everyday language is imprecise and ambiguous, so informal descriptions in English are often inadequate. The set notation used in Examples 1.9 and 1.10 is more suitable, but limited. As we proceed we will learn about several language-definition mechanisms that are useful in different circumstances. Here we introduce a common and powerful one, the notion of a **grammar**.

A grammar for the English language tells us whether a particular sentence is well-formed or not. A typical rule of English grammar is "a sentence can consist of a noun phrase followed by a predicate." More concisely we write this as

$$\langle sentence \rangle \rightarrow \langle noun\_phrase \rangle \, \langle predicate \rangle,$$

with the obvious interpretation. This is, of course, not enough to deal with actual sentences. We must now provide definitions for the newly introduced constructs $\langle noun\_phrase \rangle$ and $\langle predicate \rangle$. If we do so by

$$\langle noun\_phrase \rangle \rightarrow \langle article \rangle \, \langle noun \rangle,$$
$$\langle predicate \rangle \rightarrow \langle verb \rangle,$$

and if we associate the actual words "a" and "the" with $\langle article \rangle$, "boy" and "dog" with $\langle noun \rangle$, and "runs" and "walks" with $\langle verb \rangle$, then the grammar tells us that the sentences "a boy runs" and "the dog walks" are properly formed. If we were to give a complete grammar, then in theory, every proper sentence could be explained this way.

This example illustrates the definition of a general concept in terms of simple ones. We start with the top-level concept, here $\langle sentence \rangle$, and

مادة محمية بموجب حقوق النشر

successively reduce it to the irreducible building blocks of the language. The generalization of these ideas leads us to formal grammars.

Definition 1.1

A grammar G is defined as a quadruple

$$G = (V, T, S, P),$$

where V is a finite set of objects called **variables**,
T is a finite set of objects called **terminal symbols**,
$S \in V$ is a special symbol called the **start** variable,
P is a finite set of **productions**.

It will be assumed without further mention that the sets V and T are nonempty and disjoint.

The production rules are the heart of a grammar; they specify how the grammar transforms one string into another, and through this they define a language associated with the grammar. In our discussion we will assume that all production rules are of the form

$$x \to y,$$

where x is an element of $(V \cup T)^+$ and y is in $(V \cup T)^*$. The productions are applied in the following manner: Given a string w of the form

$$w = uxv,$$

we say the production $x \to y$ is applicable to this string, and we may use it to replace x with y, thereby obtaining a new string

$$z = uyv.$$

This is written as

$$w \Rightarrow z.$$

We say that w **derives** z or that z is derived from w. Successive strings are derived by applying the productions of the grammar in arbitrary order. A production can be used whenever it is applicable, and it can be applied as often as desired. If

$$w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n,$$

مادة محمية بموجب حقوق النشر

we say that w_1 derives w_n and write

$$w_1 \overset{*}{\Rightarrow} w_n.$$

The $*$ indicates that an unspecified number of steps (including zero) can be taken to derive w_n from w_1.

By applying the production rules in a different order, a given grammar can normally generate many strings. The set of all such terminal strings is the language defined or generated by the grammar.

Definition 1.2

Let $G = (V, T, S, P)$ be a grammar. Then the set

$$L(G) = \left\{ w \in T^* : S \overset{*}{\Rightarrow} w \right\}$$

is the language generated by G.

If $w \in L(G)$, then the sequence

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$$

is a **derivation** of the sentence w. The strings $S, w_1, w_2, ..., w_n$, which contain variables as well as terminals, are called **sentential forms** of the derivation.

Example 1.11 Consider the grammar

$$G = (\{S\}, \{a, b\}, S, P),$$

with P given by

$$S \to aSb,$$
$$S \to \lambda.$$

Then

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb,$$

so we can write

$$S \overset{*}{\Rightarrow} aabb.$$

The string $aabb$ is a sentence in the language generated by G, while $aaSbb$ is a sentential form.

مادة محمية بموجب حقوق النشر

where both w_1 and w_2 are in L. This case can be taken care of by the production $S \to SS$.

Once we see the argument intuitively, we are ready to proceed more rigorously. Again we use induction. Assume that all $w \in L$ with $|w| \leq 2n$ can be derived with G. Take any $w \in L$ of length $2n+2$. If $w = aw_1b$, then w_1 is in L, and $|w_1| = 2n$. Therefore, by assumption,

$$S \overset{*}{\Rightarrow} w_1.$$

Then

$$S \Rightarrow aSb \overset{*}{\Rightarrow} aw_1b = w$$

is possible, and w can be derived with G. Obviously, similar arguments can be made if $w = bw_1a$.

If w is not of this form, that is, if it starts and ends with the same symbol, then the counting argument tells us that it must have the form $w = w_1w_2$, with w_1 and w_2 both in L and of length less than or equal to $2n$. Hence again we see that

$$S \Rightarrow SS \overset{*}{\Rightarrow} w_1S \overset{*}{\Rightarrow} w_1w_2 = w$$

is possible.

Since the inductive assumption is clearly satisfied for $n = 1$, we have a basis, and the claim is true for all n, completing our argument.

Normally, a given language has many grammars that generate it. Even though these grammars are different, they are equivalent in some sense. We say that two grammars G_1 and G_2 are **equivalent** if they generate the same language, that is, if

$$L(G_1) = L(G_2).$$

As we will see later, it is not always easy to see if two grammars are equivalent.

Example 1.14 Consider the grammar $G_1 = (\{A, S\}, \{a, b\}, S, P_1)$, with P_1 consisting of the productions

$$S \to aAb|\lambda,$$
$$A \to aAb|\lambda.$$

Here we introduce a convenient shorthand notation in which several production rules with the same left-hand sides are written on the same line, with alternative right-hand sides separated by $|$. In this notation $S \to aAb|\lambda$ stands for the two productions $S \to aAb$ and $S \to \lambda$.

مادة محمية بموجب حقوق النشر

(c) $L_3 = \left\{a^{n+2}b^n : n \geq 1\right\}$.

(d) $L_4 = \left\{a^n b^{n-3} : n \geq 3\right\}$.

(e) $L_1 L_2$.

(f) $L_1 \cup L_2$.

(g) L_1^3.

(h) L_1^*.

(i) $L_1 - \overline{L_4}$.

★15. Find grammars for the following languages on $\Sigma = \{a\}$.

(a) $L = \{w : |w| \mod 3 = 0\}$.

(b) $L = \{w : |w| \mod 3 > 0\}$.

(c) $L = \{w : |w| \mod 3 \neq |w| \mod 2\}$.

(d) $L = \{w : |w| \mod 3 \geq |w| \mod 2\}$.

16. Find a grammar that generates the language

$$L = \left\{ww^R : w \in \{a, b\}^+\right\}.$$

Give a complete justification for your answer.

17. Give a verbal description of the language generated by

$$S \rightarrow aSb|bSa|a.$$

18. Using the notation of Example 1.13, find grammars for the languages below. Assume $\Sigma = \{a, b\}$.

(a) $L = \{w : n_a(w) = n_b(w) + 1\}$.

(b) $L = \{w : n_a(w) > n_b(w)\}$.

★ (c) $L = \{w : n_a(w) = 2n_b(w)\}$.

(d) $L = \{w \in \{a, b\}^* : |n_a(w) - n_b(w)| = 1\}$.

19. Repeat the previous exercise with $\Sigma = \{a, b, c\}$.

20. Complete the arguments in Example 1.14, showing that $L(G_1)$ does in fact generate the given language.

21. Are the two grammars with respective productions

$$S \rightarrow aSb\,|ab|\,\lambda,$$

and

$$\begin{aligned} S &\rightarrow aAb|ab, \\ A &\rightarrow aAb|\lambda \end{aligned}$$

equivalent? Assume that S is the start symbol in both cases.

مادة محمية بموجب حقوق النشر

22. Show that the grammar $G = (\{S\}, \{a, b\}, S, P)$, with productions

$$S \to SS \,|\, SSS \,|\, aSb \,|\, bSa \,|\, \lambda,$$

is equivalent to the grammar in Example 1.13.

23. Show that the grammars

$$S \to aSb|bSa|SS|a$$

and

$$S \to aSb|bSa|a$$

are not equivalent.

1.3 Some Applications*

Although we stress the abstract and mathematical nature of formal languages and automata, it turns out that these concepts have widespread applications in computer science and are, in fact, a common theme that connects many specialty areas. In this section, we present some simple examples to give the reader some assurance that what we study here is not just a collection of abstractions, but is something that helps us understand many important, real problems.

Formal languages and grammars are used widely in connection with programming languages. In most of our programming, we work with a more or less intuitive understanding of the language in which we write. Occasionally though, when using an unfamiliar feature, we may need to refer to precise descriptions such as the syntax diagrams found in most programming texts. If we write a compiler, or if we wish to reason about the correctness of a program, a precise description of the language is needed at almost every step. Among the ways in which programming languages can be defined precisely, grammars are perhaps the most widely used.

The grammars that describe a typical language like Pascal or C are very extensive. For an example, let us take a smaller language that is part of a larger one.

Example 1.15 The rules for variable identifiers in C are

1. An identifier is a sequence of letters, digits, and underscores.
2. An identifier must start with a letter or an underscore.
3. Identifiers allow upper- and lower-case letters.

مادة محمية بموجب حقوق النشر

Formally, these rules can be described by a grammar.

$< id > \rightarrow < letter >< rest > | < undrscr >< rest >$
$< rest > \rightarrow < letter >< rest > | < digit >< rest > | < undrscr >< rest > |\lambda$
$< letter > \rightarrow a|b|...|z|A|B|...|Z$
$< digit > \rightarrow 0|1|...|9$
$< undrscr > \rightarrow$ -

In this grammar, the variables are $\langle id \rangle$, $\langle letter \rangle$, $\langle digit \rangle$, $\langle undrscr \rangle$, and $\langle rest \rangle$. The letters, digits, and the underscore are terminals. A derivation of $a0$ is

$$\begin{aligned} \langle id \rangle &\Rightarrow \langle letter \rangle \langle rest \rangle \\ &\Rightarrow a \langle rest \rangle \\ &\Rightarrow a \langle digit \rangle \langle rest \rangle \\ &\Rightarrow a0 \langle rest \rangle \\ &\Rightarrow a0. \end{aligned}$$

The definition of programming languages through grammars is common and very useful. But there are alternatives that are often convenient. For example, we can describe a language by an accepter, taking every string that is accepted as part of the language. To talk about this in a precise way, we will need to give a more formal definition of an automaton. We will do this shortly; for the moment, let us proceed in a more intuitive way.

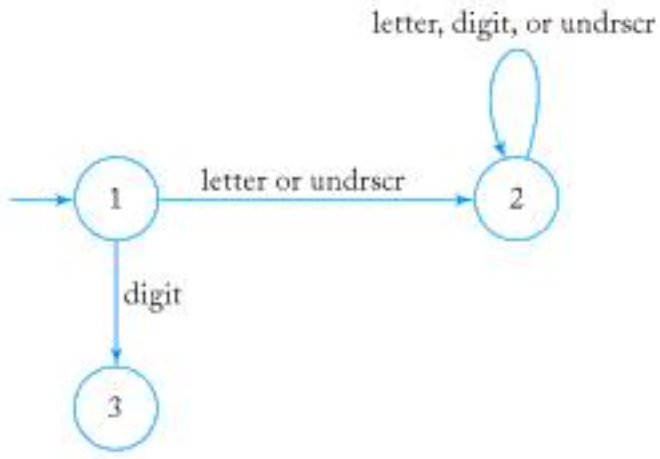
An automaton can be represented by a graph in which the vertices give the internal states and the edges transitions. The labels on the edges show what happens (in terms of input and output) during the transition. For example, Figure 1.5 represents a transition from State 1 to State 2, which is taken when the input symbol is a. With this intuitive picture in mind, let us look at another way of describing C identifiers.

Figure 1.5

Example 1.16 Figure 1.6 is an automaton that accepts all legal C identifiers. Some interpretation is necessary. We assume that initially the automaton is in State 1; we indicate this by drawing an arrow (not originating in any vertex) to this state. As always, the string to be examined is read left to right, one character at each step. When the first symbol is a letter or an underscore, the automaton goes into State 2, after which the rest of the string is immaterial. State 2 therefore represents the "yes" state of the accepter. Conversely, if

مادة محمية بموجب حقوق النشر

Figure 1.6



the first symbol is a digit, the automaton will go into State 3, the "no" state, and remain there. In our solution, we assume that no input other than letters, digits, or underscores is possible.

Compilers and other translators that convert a program from one language to another make extensive use of the ideas touched on in these examples. Programming languages can be defined precisely through grammars, as in Example 1.15, and both grammars and automata play a fundamental role in the decision processes by which a specific piece of code is accepted as satisfying the conditions of a programming language. The above example gives a first hint of how this is done; subsequent examples will expand on this observation.

Transducers will be discussed briefly in Appendix A; the following example previews this subject.

Example 1.17 A binary adder is an integral part of any general-purpose computer. Such an adder takes two bit strings representing numbers and produces their sum as output. For simplicity, let us assume that we are dealing only with positive integers and that we use a representation in which

$$x = a_0 a_1 \cdots a_n$$

stands for the integer

$$v(x) = \sum_{i=0}^{n} a_i 2^i.$$

This is the usual binary representation in reverse.

A serial adder processes two such numbers $x = a_0 a_1 \cdots a_n$, and $y = b_0 b_1 \cdots b_n$, bit by bit, starting at the left end. Each bit addition creates a

مادة محمية بموجب حقوق النشر

Figure 1.7

a_i \ b_i	0	1
0	0 No carry	1 No carry
1	1 No carry	0 Carry

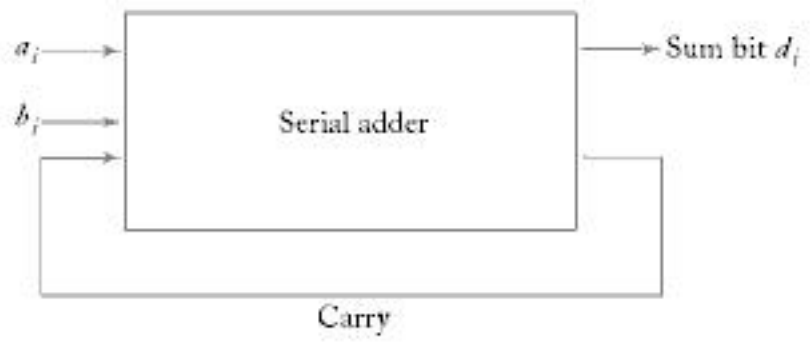
digit for the sum as well as a carry digit for the next higher position. A binary addition table (Figure 1.7) summarizes the process.

A block diagram of the kind we saw when we first studied computers is given in Figure 1.8. It tells us that an adder is a box that accepts two bits and produces their sum bit and a possible carry. It describes what an adder does, but explains little about its internal workings. An automaton (now a transducer) can make this much more explicit.

The input to the transducer are the bit pairs (a_i, b_i), the output will be the sum bit d_i. Again, we represent the automaton by a graph now labeling the edges $(a_i, b_i)\,/d_i$. The carry from one step to the next is remembered by the automaton via two internal states labeled "carry" and "no carry." Initially, the transducer will be in state "no carry." It will remain in this state until a bit pair $(1, 1)$ is encountered; this will generate a carry that takes the automaton into the "carry" state. The presence of a carry is then taken into account when the next bit pair is read. A complete picture of a serial adder is given in Figure 1.9. Follow this through with a few examples to convince yourself that it works correctly.

As this example indicates, the automaton serves as a bridge between the very high-level, functional description of a circuit and its logical implementation through transistors, gates, and flip-flops. The automaton clearly shows the decision logic, yet it is formal enough to lend itself to precise

Figure 1.8



مادة محمية بموجب حقوق النشر

Chapter 2

Finite Automata

Our introduction in the first chapter to the basic concepts of computation, particularly the discussion of automata, is brief and informal. At this point, we have only a general understanding of what an automaton is and how it can be represented by a graph. To progress, we must be more precise, provide formal definitions, and start to develop rigorous results. We begin with finite accepters, which are a simple, special case of the general scheme introduced in the last chapter. This type of automaton is characterized by having no temporary storage. Since an input file cannot be rewritten, a finite automaton is severely limited in its capacity to "remember" things during the computation. A finite amount of information can be retained in the control unit by placing the unit into a specific state. But since the number of such states is finite, a finite automaton can only deal with situations in which the information to be stored at any time is strictly bounded. The automaton in Example 1.16 is an instance of a finite accepter.

مادة محمية بموجب حقوق النشر

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A dfa will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the dfa stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Example 2.2 Consider the dfa in Figure 2.2.

In drawing Figure 2.2 we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: The transition is taken whenever the input symbol matches any of the edge labels.

The automaton in Figure 2.2 remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the dfa goes into state q_2, from which it can never escape. The state q_2 is a **trap state**. We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

$$L = \{a^n b : n \geq 0\}.$$

Figure 2.2

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (2.1) and (2.2), the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must, of course, have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

مادة محمية بموجب حقوق النشر

deterministic finite accepters is quite limited. The structure and properties of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.3

A language L is called **regular** if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

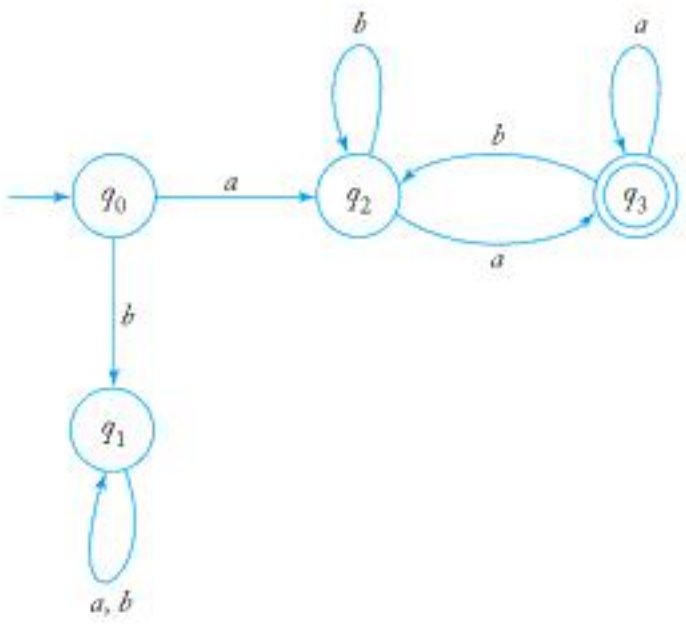
Example 2.5 Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

is regular.

To show that this or any other language is regular, all we have to do is find a dfa for it. The construction of a dfa for this language is similar to Example 2.3, but a little more complicated. What this dfa must do is check whether a string begins and ends with an a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string. This difficulty is overcome by simply putting the dfa into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the dfa out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in Figure 2.6. Again,

Figure 2.6



مادة محمية بموجب حقوق النشر

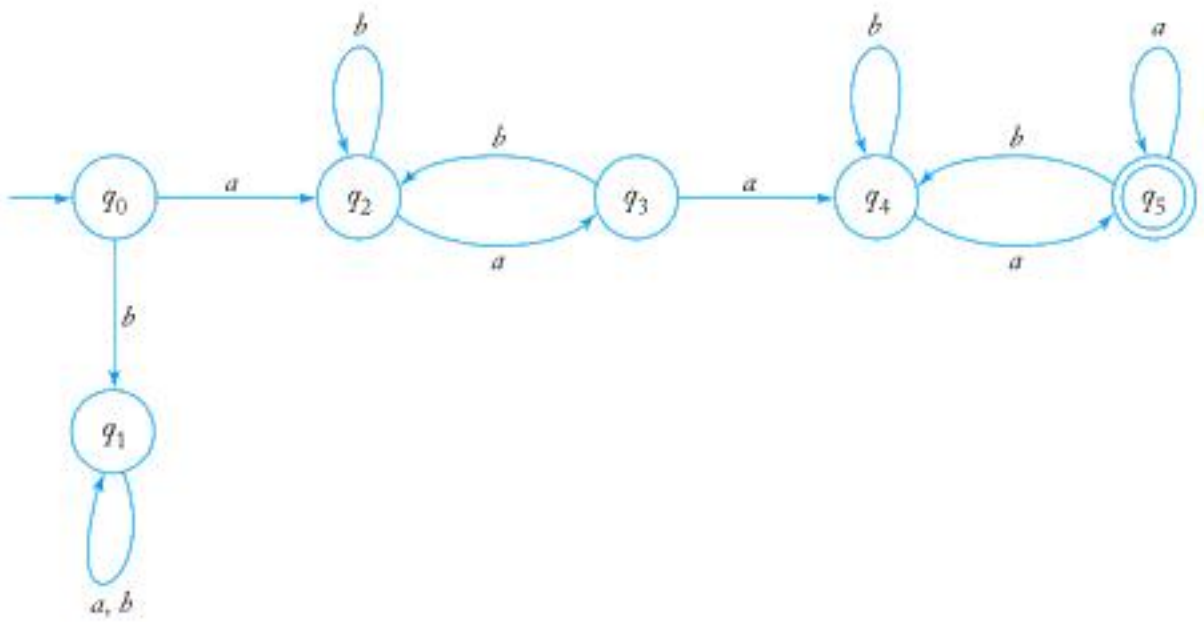
trace a few examples to see why this works. After one or two tests, it will be obvious that the dfa accepts a string if and only if it begins and ends with an a. Since we have constructed a dfa for the language, we can claim that, by definition, the language is regular.

Example 2.6 Let L be the language in Example 2.5. Show that L^2 is regular. Again we show that the language is regular by constructing a dfa for it. We can write an explicit expression for L^2, namely,

$$L^2 = \left\{aw_1aaaw_2a : w_1, w_2 \in \{a, b\}^*\right\}.$$

Therefore, we need a dfa that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram in Figure 2.6 can be used as a starting point, but the vertex q_3 has to be modified. This state can no longer be final since, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part. We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in Figure 2.7. This dfa accepts L^2, which is therefore regular.

Figure 2.7



The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We will see later that this is indeed correct.

مادة محمية بموجب حقوق النشر

EXERCISES

1. Which of the strings 0001, 01001, 0000110 are accepted by the dfa in Figure 2.1?
2. For $\Sigma = \{a, b\}$, construct dfa's that accept the sets consisting of
   (a) all strings with exactly one a,
   (b) all strings with at least one a,
   (c) all strings with no more than three a's,
   (d) all strings with at least one a and exactly two b's,
   (e) all the strings with exactly two a's and more than two b's.
3. Show that if we change Figure 2.6, making q_3 a nonfinal state and making q_0, q_1, q_2 final states, the resulting dfa accepts $\overline{L}$.
4. Generalize the observation in the previous exercise. Specifically, show that if $M = (Q, \Sigma, \delta, q_0, F)$ and $\widehat{M} = (Q, \Sigma, \delta, q_0, Q - F)$ are two dfa's, then $\overline{L(M)} = L\left(\widehat{M}\right)$.
5. Give dfa's for the languages
   (a) $L = \left\{ab^5wb^2 : w \in \{a, b\}^*\right\}$,
   (b) $L = \{ab^n a^m : n \geq 2, m \geq 3\}$,
   (c) $L = \{w_1 abw_2 : w_1 \in \{a, b\}^*, w_2 \in \{a, b\}^*\}$,
   (d) $L = \{ba^n : n \geq 1, n \neq 5\}$.
6. With $\Sigma = \{a, b\}$, give a dfa for $L = \{w_1 aw_2 : |w_1| \geq 3, |w_2| \leq 5\}$.
7. Find dfa's for the following languages on $\Sigma = \{a, b\}$.
   (a) $L = \{w : |w| \mod 3 = 0\}$.
   (b) $L = \{w : |w| \mod 5 \neq 0\}$.
   (c) $L = \{w : n_a(w) \mod 3 > 1\}$.
   (d) $L = \{w : n_a(w) \mod 3 > n_b(w) \mod 3\}$.
   (e) $L = \{w : (n_a(w) - n_b(w)) \mod 3 > 0\}$.
   (f) $L = \{w : (n_a(w) + 2n_b(w)) \mod 3 < 2\}$.
   (g) $L = \{w : |w| \mod 3 = 0, |w| \neq 6\}$.

★8. A run in a string is a substring of length at least two, as long as possible and consisting entirely of the same symbol. For instance, the string *abbbaab* contains a run of b's of length three and a run of a's of length two. Find dfa's for the following languages on $\{a, b\}$.
   (a) $L = \{w : w$ contains no runs of length less than four$\}$.
   (b) $L = \{w :$ every run of a's has length either two or three$\}$.

مادة محمية بموجب حقوق النشر

(c) $L = \{w:$ there are at most two runs of a's of length three$\}$.

(d) $L = \{w:$ there are exactly two runs of a's of length 3$\}$.

9. Consider the set of strings on $\{0,1\}$ defined by the requirements below. For each, construct an accepting dfa.

   (a) Every 00 is followed immediately by a 1. For example, the strings 101, 0010, 0010011001 are in the language, but 0001 and 00100 are not.

   (b) All strings containing 00 but not 000.

   (c) The leftmost symbol differs from the rightmost one.

   (d) Every substring of four symbols has at most two 0's. For example, 001110 and 011001 are in the language, but 10010 is not since one of its substrings, 0010, contains three zeros.

   (e) All strings of length five or more in which the fourth symbol from the right end is different from the leftmost symbol.

   (f) All strings in which the leftmost two symbols and the rightmost two symbols are identical.

   (g) All strings of length four or greater in which the leftmost three symbols are the same, but different from the rightmost symbol.

★ 10. Construct a dfa that accepts strings on $\{0,1\}$ if and only if the value of the string, interpreted as a binary representation of an integer, is zero modulo five. For example, 0101 and 1111, representing the integers 5 and 15, respectively, are to be accepted.

11. Show that the language $L = \{vwv : v, w \in \{a,b\}^*, |v| = 2\}$ is regular.

12. Show that $L = \{a^n : n \geq 4\}$ is regular.

13. Show that the language $L = \{a^n : n \geq 0, n \neq 4\}$ is regular.

14. Show that the language $L = \{a^n : n$ is either a multiple of three or a multiple of 5$\}$ is regular.

15. Show that the language $L = \{a^n : n$ is a multiple of three, but not a multiple of 5$\}$ is regular.

16. Show that the set of all real numbers in C is a regular language.

17. Show that if L is regular, so is $L - \{\lambda\}$.

18. Show that if L is regular, so is $L \cup \{a\}$, for all $a \in \Sigma$.

19. Use (2.1) and (2.2) to show that

$$\delta^*(q, wv) = \delta^*(\delta^*(q, w), v)$$

for all $w, v \in \Sigma^*$.

20. Let L be the language accepted by the automaton in Figure 2.2. Find a dfa that accepts L^2.

21. Let L be the language accepted by the automaton in Figure 2.2. Find a dfa for the language $L^2 - L$.

مادة محمية بموجب حقوق النشر

22. Let L be the language in Example 2.5. Show that L^* is regular.

23. Let G_M be the transition graph for some dfa M. Prove the following.

    (a) If $L(M)$ is infinite, then G_M must have at least one cycle for which there is a path from the initial vertex to some vertex in the cycle and a path from some vertex in the cycle to some final vertex.

    (b) If $L(M)$ is finite, then no such cycle exists.

24. Let us define an operation *truncate*, which removes the rightmost symbol from any string. For example, $truncate(aaaba)$ is $aaab$. The operation can be extended to languages by

$$truncate(L) = \{truncate(w) : w \in L\}.$$

    Show how, given a dfa for any regular language L, one can construct a dfa for $truncate(L)$. From this, prove that if L is a regular language not containing λ, then $truncate(L)$ is also regular.

25. While the language accepted by a given dfa is unique, there are normally many dfa's that accept a language. Find a dfa with exactly six states that accepts the same language as the dfa in Figure 2.4.

26. Can you find a dfa with three states that accepts the language of the dfa in Figure 2.4? If not, can you give convincing arguments that no such dfa can exist?

2.2 Nondeterministic Finite Accepters

Finite accepters are more complicated if we allow them to act nondeterministically. Nondeterminism is a powerful but, at first sight, unusual idea. We normally think of computers as completely deterministic, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful notion, as we shall see as we proceed.

Definition of a Nondeterministic Accepter

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

Definition 2.4

A **nondeterministic finite accepter** or **nfa** is defined by the quintuple

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

مادة محمية بموجب حقوق النشر

Regular Expressions for Describing Simple Patterns

In Example 1.15 and in Exercise 16, Section 2.1, we explored the connection between finite accepters and some of the simpler constituents of programming languages, such as identifiers, or integers and real numbers. The relation between finite automata and regular expressions means that we can also use regular expressions as a way of describing these features. This is easy to see; for example, in many programming languages the set of integer constants is defined by the regular expression

$$sdd^*,$$

where s stands for the sign, with possible values from $\{+, -, \lambda\}$, and d stands for the digits 0 to 9. Integer constants are a simple case of what is sometimes called a "pattern," a term that refers to a set of objects having some common properties. Pattern matching refers to assigning a given object to one of several categories. Often, the key to successful pattern matching is finding an effective way to describe the patterns. This is a complicated and extensive area of computer science to which we can only briefly allude. The following example is a simplified, but nevertheless instructive, demonstration of how the ideas we have talked about so far have been found useful in pattern matching.

Example 3.12 An application of pattern matching occurs in text editing. All text editors allow files to be scanned for the occurrence of a given string; most editors extend this to permit searching for patterns. For example, the *vi* editor in the UNIX operating system recognizes the command $/aba^*c/$ as an instruction to search the file for the first occurrence of the string ab, followed by an arbitrary number of a's, followed by a c. We see from this example the need for pattern-matching editors to work with regular expressions.

A challenging task in such an application is to write an efficient program for recognizing string patterns. Searching a file for occurrences of a given string is a very simple programming exercise, but here the situation is more complicated. We have to deal with an unlimited number of arbitrarily complicated patterns; furthermore, the patterns are not fixed beforehand, but created at run time. The pattern description is part of the input, so the recognition process must be flexible. To solve this problem, ideas from automata theory are often used.

If the pattern is specified by a regular expression, the pattern-recognition program can take this description and convert it into an equivalent nfa using the construction in Theorem 3.1. Theorem 2.2 may then be used to reduce this to a dfa. This dfa, in the form of a transition table, is effectively the pattern-matching algorithm. All the programmer has to do is to provide a driver that gives the general framework for using the table. In this way

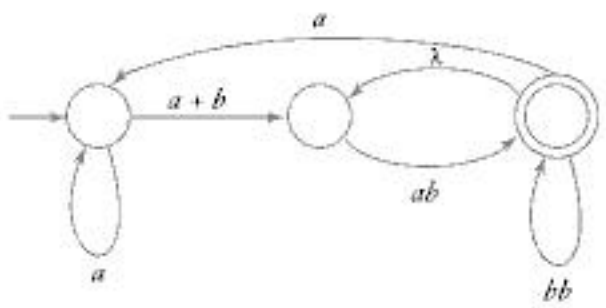
مادة محمية بموجب حقوق النشر

we can automatically handle a large number of patterns that are defined at run time.

The efficiency of the program must also be considered. The construction of finite automata from regular expressions using Theorems 2.1 and 3.1 tends to yield automata with many states. If memory space is a problem, the state reduction method described in Section 2.4 is helpful.

Exercises

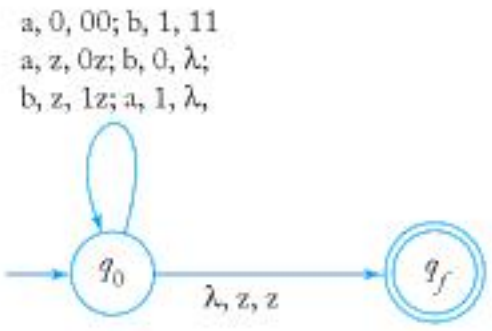
1. Use the construction in Theorem 3.1 to find an nfa that accepts the language $L\left(ab^*aa + bba^*ab\right)$.
2. Find an nfa that accepts the complement of the language in Exercise 1.
3. Give an nfa that accepts the language $L\left((a+b)^*\, b\,(a+bb)^*\right)$.
4. Find dfa's that accept the following languages.
   (a) $L\left(aa^* + aba^*b^*\right)$.
   (b) $L\left(ab\,(a+ab)^*\,(a+aa)\right)$.
   (c) $L\left((abab)^* + (aaa^* + b)^*\right)$.
   (d) $L\left(((aa^*)^*\, b)^*\right)$.
5. Find dfa's that accept the following languages.
   (a) $L = L\left(ab^*a^*\right) \cup L\left((ab)^*\, ba\right)$.
   (b) $L = L\left(ab^*a^*\right) \cap L\left((ab)^*\, ba\right)$.
6. Find an nfa for Exercise 17(f), Section 3.1. Use this to derive a regular expression for that language.
7. Find the minimal dfa that accepts $L(a^*bb) \cup L(ab^*ba)$.
8. Consider the following generalized transition graph.



   (a) Find an equivalent generalized transition graph with only two states.
   (b) What is the language accepted by this graph?

مادة محمية بموجب حقوق النشر

Figure 7.3



In processing the string *baab*, the npda makes the moves

$$\begin{aligned}(q_0, baab, z) &\vdash (q_0, aab, 1z) \vdash (q_0, ab, z)\\ &\vdash (q_0, b, 0z) \vdash (q_0, \lambda, z) \vdash (q_f, \lambda, z)\end{aligned}$$

and hence the string is accepted.

Example 7.5 To construct an npda for accepting the language

$$L = \left\{ww^R : w \in \{a, b\}^+\right\},$$

we use the fact that the symbols are retrieved from a stack in the reverse order of their insertion. When reading the first part of the string, we push consecutive symbols on the stack. For the second part, we compare the current input symbol with the top of the stack, continuing as long as the two match. Since symbols are retrieved from the stack in reverse of the order in which they were inserted, a complete match will be achieved if and only if the input is of the form ww^R.

An apparent difficulty with this suggestion is that we do not know the middle of the string, that is, where w ends and w^R starts. But the nondeterministic nature of the automaton helps us with this; the npda correctly guesses where the middle is and switches states at that point. A solution to the problem is given by $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$, where

$$\begin{aligned}Q &= \{q_0, q_1, q_2\},\\ \Sigma &= \{a, b\},\\ \Gamma &= \{a, b, z\},\\ F &= \{q_2\}.\end{aligned}$$

مادة محمية بموجب حقوق النشر

مادة محمية بموجب حقوق النشر

مادة محمية بموجب حقوق النشر