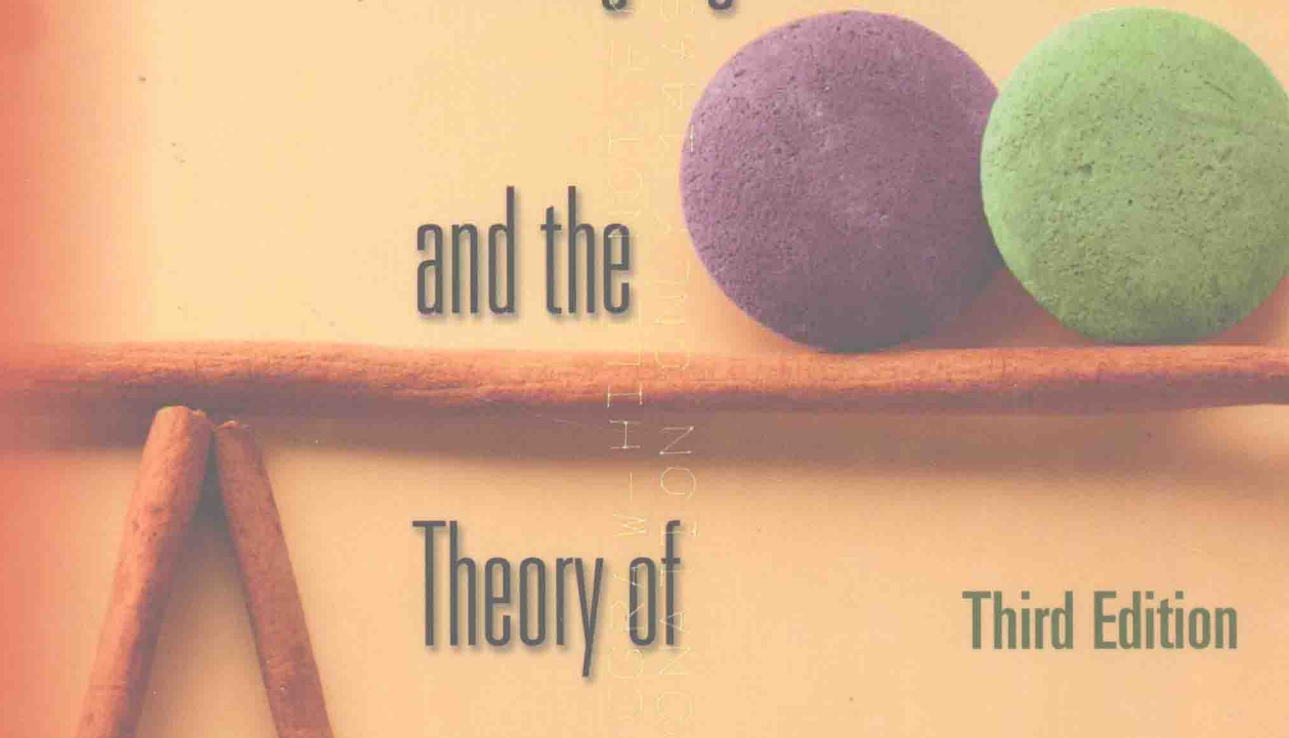
Introduction to Languages and the Theory of Computation

John Martin

Third Edition



Introduction to Languages and The Theory of Computation

Third Edition

John C. Martin
North Dakota State University

Mc Graw Hill

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

McGraw-Hill Higher Education

A Division of The ***McGraw-Hill*** *Companies*

INTRODUCTION TO LANGUAGES AND THE THEORY OF COMPUTATION
THIRD EDITION

International 1 2 3 4 5 6 7 8 9 0 QPF/QPF 0 9 8 7 6 5 4 3 2
Domestic 3 4 5 6 7 8 9 0 QPF/QPF 0 9 8 7 6 5

ISBN-13: 978-0-07-232200-2 ISBN-13: 978-0-07-119854-7 (ISE)
ISBN-10: 0-07-232200-4 ISBN-10: 0-07-119854-7

Publisher: *Elizabeth A. Jones*
Developmental editor: *Melinda Dougharty*
Executive marketing manager: *John Wannemacher*
Lead project manager: *Peggy J. Selle*
Lead production supervisor: *Sandy Ludovissy*
Lead media project manager: *Audrey A. Reiter*
Designer: *K. Wayne Harms*
Cover/interior designer: *Rokusek Design*
Cover image: *Ryoichi Utsumi (IMA) Photonica*
Compositor: *Techsetters*
Typeface: *10/12 Times Roman*
Printer: *Quebecor World Fairfield, PA*

Library of Congress Cataloging-in-Publication Data

Martin, John C.
Introduction to languages and the theory of computation / John Martin.—3rd ed.
p. cm.
Includes index.
ISBN 0-07-232200-4—ISBN 0-07-119854-7
1. Sequential machine theory. 2. Computable functions. I. Title.

QA267.5.S4 M29 2003
511.3—dc21 2002070865
CIP

www.mhhe.com

TO PIPPA

ABOUT THE AUTHOR

John C. Martin attended Rice University both as an undergraduate and as a graduate student, receiving a B.A. in mathematics in 1966 and a Ph.D. in 1971. He taught for two years at the University of Hawaii in Honolulu before joining the faculty of North Dakota State University, where he is an associate professor of computer science.

PREFACE

This book is an introduction to the theory of computation. It emphasizes formal languages, models of computation, and computability, and it includes an introduction to computational complexity and *NP*-completeness.

Most students studying these topics have already had experience in the *practice* of computation. They have used a number of technologies related to computers; now they can begin to acquire an appreciation of computer science as a coherent discipline. The ideas are profound—and fun to think about—and the principles will not quickly become obsolete. Finally, students can gain proficiency with mathematical tools and formal methods, at the same time that they see how these techniques are applied to computing.

I believe that the best way to present theoretical topics such as the ones in this book is to take advantage of the clarity and precision of mathematical language—provided the presentation is accessible to readers who are still learning to use this language. The book attempts to introduce the necessary mathematical tools gently and gradually, in the context in which they are used, and to provide discussion and examples that make the language intelligible. The first two chapters present the topics from discrete mathematics that come up later, including a detailed discussion of mathematical induction. As a result, the text can be read by students without a strong background in discrete math, and it should also be appropriate for students whose skills in that area need to be consolidated and sharpened.

The organizational changes in the third edition are not as dramatic as those in the second. One chapter was broken up and distributed among the remaining fourteen, and sections of several chapters were reworked and rearranged. In addition to changes in organization, there were plenty of opportunities throughout to rewrite, to correct proofs and examples and make them easier to understand, to add examples, and to replace examples by others that illustrate principles better. Some exercises have been added, some others have been modified, and the exercises in each chapter have been grouped into ordinary ones and more challenging ones. In the Turing machine chapter, I have followed the advice of two reviewers in adopting a more standard and more intuitive definition of *halting*.

Whether or not Part I is covered in detail, I recommend covering Section 1.5, which introduces notation and terminology involving languages. It may also be desirable to review mathematical induction, particularly the sections on recursive definitions and structural induction and the examples having to do with formal languages. At North Dakota State, the text is used in a two-semester sequence required of undergraduate computer science majors, and there is more than enough material for both semesters. A one-semester course omitting most of Part I could cover regular and context-free languages, and the corresponding automata, and at least some of the theory of Turing machines and solvability. In addition, since most of Parts IV, V, and

VI are substantially independent of the first three parts, the text can also be used in a course on Turing machines, computability, and complexity.

I am grateful to the many people who have helped me with all three editions of this text. Particular thanks are due to Ting-Lu Huang, who pointed out an error in the proof of Theorem 4.2 in the second edition, and to Jonathan Goldstine, who provided several corrections to Chapters 7 and 8. I appreciate the thoughtful and detailed comments of Bruce Wieand, North Carolina State University; Edward Ashcroft, Arizona State University; Ding-Zhu Du, University of Minnesota; William D. Shoaff, Florida Institute of Technology; and Sharon Tuttle, Humboldt State University, who reviewed the second edition, and Ding-Zhu Du, University of Minnesota; Leonard M. Faltz, Arizona State University; and Nilfur Onder, Michigan Tech, who reviewed a preliminary version of this edition. Their help has resulted in a number of improvements, including the modification in Chapter 9 mentioned earlier. Melinda Dougharty at McGraw-Hill has been delightful to work with, and I also appreciate the support and professionalism of Betsy Jones and Peggy Selle. Finally, thanks once again to my wife Pippa for her help, both tangible and intangible.

John C. Martin

INTRODUCTION

In order to study the theory of computation, let us try to say what a computation is. We might say that it consists of executing an *algorithm*: starting with some input and following a step-by-step procedure that will produce a result. Exactly what kinds of steps are allowed in an algorithm? One approach is to think about the steps allowed in high-level languages that are used to program computers (C, for example). Instead, however, we will think about the computers themselves. We will say that a step will be permitted in a computation if it is an operation the computer can make. In other words, a computation is simply a sequence of steps that can be performed by a computer! We will be able to talk precisely about algorithms and computations once we know precisely what kinds of computers we will study.

The computers will not be *actual* computers. In the first place, a theory based on the specifications of an actual piece of hardware would not be very useful, because it would have to be changed every time the hardware was changed or enhanced. Even more importantly, actual computers are much too complicated; the idealized computers we will study are simple. We will study several *abstract machines*, or models of computation, which will be defined mathematically. Some of them are as powerful in principle as real computers (or even more so, because they are not subject to physical constraints on memory), while the simpler ones are less powerful. These simpler machines are still worth studying, because they make it easier to introduce some of the mathematical formalisms we will use in our theory and because the computations they can carry out are performed by real-world counterparts in many real-world situations.

We can understand the "languages" part of the subject by considering the idea of a *decision problem*, a computational problem for which every specific instance can be answered "yes" or "no." A familiar numerical example is the problem: Given a positive integer n, is it prime? The number n is encoded as a string of digits, and a computation that solves the problem starts with this input string. We can think about this as a *language recognition* problem: to take an arbitrary string of digits and determine whether it is one of the strings in the language of all strings representing primes. In the same way, solving any decision problem can be thought of as recognizing a certain language, the language of all strings representing instances of the problem for which the answer is "yes." Not all computational problems are decision problems, and the more powerful of our models of computation will allow us to handle more general kinds; however, even a more general problem can often be approached by considering a comparable decision problem. For example, if f is a function, being able to answer the question: given x and y, is $y = f(x)$? is tantamount to being able to compute $f(x)$ for an arbitrary x. The problem of language recognition will be a unifying theme in our discussion of abstract models of computation. Computing machines of different types can recognize languages of different complexity, and the various computation models will result in a corresponding hierarchy of language types.

The simplest type of abstract machine we consider is a *finite automaton*, or finite-state machine. The underlying principle is a very general one. Any system that is at each moment in one of a finite number of discrete states, and moves among these states in a predictable way in response to individual input signals, can be modeled by a finite automaton. The languages these machines can recognize are the *regular* languages, which can also be described as the ones obtained from one-element languages by repeated applications of certain basic operations. Regular languages include some that arise naturally as "pieces" of programming languages. The corresponding machines in software form have been applied to various problems in compiler design and text editing, among others.

The most obvious limitation of a finite automaton is that, except for being able to keep track of its current state, it has no memory. As you might expect, such a machine can recognize only simple languages. *Context-free* languages allow richer syntax than regular languages. They can be generated using context-free *grammars*, and they can be recognized by computing devices called *pushdown automata* (a pushdown automaton is a finite automaton with an auxiliary memory in the form of a stack). Context-free grammars were used originally to model properties of natural languages like English, which they can do only to a limited extent. They are important in computer science because they can describe much of the syntax of high-level programming languages and other related formal languages. The corresponding machines, pushdown automata, provide a natural way to approach the problem of *parsing* a statement in a high-level programming language: determining the syntax of the statement by reconstructing the sequence of rules by which it is derived in the context-free grammar.

Although the auxiliary memory makes a pushdown automaton a more powerful computing device than a finite automaton, the stack organization imposes constraints that keep the machine from being a general model of computation. A *Turing machine*, named for the English mathematician who invented it, is an even more powerful computer, and there is general agreement that such a machine is able to carry out any "step-by-step procedure" whatsoever. The languages that can be recognized by Turing machines are more general than context-free languages, and they can be produced by more general grammars. Moreover, since a Turing machine can print output strings as well as just answering yes or no, there is in principle nothing to stop such a machine from performing any computation that a full-fledged computer can, except that it is likely to do it more clumsily and less efficiently.

Nevertheless, there are limits to what a Turing machine can do; since we can describe this abstract model precisely, we can formulate specific computational problems that it cannot solve. At this point we no longer have the option of just coming up with a more powerful machine—there *are* no more powerful machines! The existence of these *unsolvable* problems means that the theory of computation is inevitably about the limitations of computers as well as their capabilities.

Finally, although a Turing machine is clumsy in the way it carries out computations, it is an effective yardstick for comparing the inherent complexity of one computational problem to that of another. Some problems that are solvable in principle are not really solvable in practice, because their solution would require impossible amounts of time and space. A simple criterion involving Turing machines is generally used to distinguish the tractable problems from the intractable ones. Although the criterion is simple, however, it is not always easy to decide which problems satisfy it. In the last chapter we discuss an interesting class of problems, those for which no one has found either a good algorithm or a convincing proof that none exists.

People have been able to compute for many thousands of years, but only very recently have people made machines that can, and *computation* as a pervasive part of our lives is an even more recent phenomenon. The theory of computation is slightly older than the electronic computer, because some of the pioneers in the field, Turing and others, were perceptive enough to anticipate the potential power of computers; their work provided the conceptual model on which the modern digital computer is based. The theory of computation has also drawn from other areas: mathematics, philosophy, linguistics, biology, and electrical engineering, to name a few. Remarkably, these elements fit together into a coherent, even elegant, theory, which has the additional advantage that it is useful and provides insight into many areas of computer science.

CONTENTS

PART 1

Mathematical Notation and Techniques

This textbook starts by reviewing some of the most fundamental mathematical ideas: sets, functions, relations, and basic principles of logic. Later in the book we will study abstract "machines"; the components of an abstract machine are sets, and the way the machine works is described by a function from one set to another. In the last section of Chapter 1, we introduce languages, which are merely sets whose elements are strings of symbols. The notation introduced in this section will be useful later, as we study classes of languages and the corresponding types of abstract machines.

Reasoning about mathematical objects involves the idea of a proof, and this is the subject of Chapter 2. The emphasis is on one particular proof technique—the principle of mathematical induction—which will be particularly useful to us in this book. A closely related idea is that of an inductive, or recursive, definition. Definitions of this type will make it easy to define languages and to establish properties of the languages using mathematical induction. ■

CHAPTER

1 Basic Mathematical Objects

1.1 | SETS

A set is determined by its elements. An easy way to describe or specify a finite set is to list all its elements. For example,

$$A = \{11, 12, 21, 22\}$$

When we enumerate a set this way, the order in which we write the elements is irrelevant. The set A could just as well be written $\{11, 21, 22, 12\}$. Writing an element more than once does not change the set: The sets $\{11, 21, 22, 11, 12, 21\}$ and $\{11, 21, 22, 12\}$ are the same.

Even if a set is infinite, it may be possible to start listing the elements in a way that makes it clear what they are. For example,

$$B = \{3, 5, 7, 9, \ldots\}$$

describes the set of odd integers greater than or equal to 3. However, although this way of describing a set is common, it is not always foolproof. Does $\{3, 5, 7, \ldots\}$ represent the same set, or does it represent the set of odd primes, or perhaps the set of integers bigger than 1 whose names contain the letter "e"?

A precise way of describing a set without listing the elements explicitly is to give a property that characterizes the elements. For example, we might write

$$B = \{x \mid x \text{ is an odd integer greater than } 1\}$$

or

$$A = \{x \mid x \text{ is a two-digit integer, each of whose digits is 1 or 2}\}$$

The notation "$\{x|$" at the beginning of both formulas is usually read "the set of all x such that."

To say that x is an element of the set A, we write

$$x \in A$$

Using this notation we might describe the set $C = \{3, 5, 7, 9, 11\}$ by writing

$$C = \{x \mid x \in B \text{ and } x \leq 11\}$$

A common way to shorten this slightly is to write

$$C = \{x \in B \mid x \leq 11\}$$

which we read "the set of x in B such that $x \leq 11$."

It is also customary to extend the notation in a different way. It would be reasonable to describe the set

$$D = \{x \mid \text{ there exist integers } i \text{ and } j\text{, both} \geq 0\text{, with } x = 3i + 7j\}$$

as "the set of numbers $3i + 7j$, where i and j are both nonnegative integers," and a concise way to write this is

$$D = \{3i + 7j \mid i, j \text{ are nonnegative integers}\}$$

Once we define $\mathcal{N}$ to be the set of nonnegative integers, or *natural numbers*, we can describe D even more concisely by writing

$$D = \{3i + 7j \mid i,\ j \in \mathcal{N}\}$$

For two sets A and B, we say that A is a *subset* of B, and write $A \subseteq B$, if every element of A is an element of B. Because a set is determined by its elements, two sets are equal if they have exactly the same elements, and this is the same as saying that each is a subset of the other. When we want to prove that $A = B$, we will need to show both statements: that $A \subseteq B$ and that $B \subseteq A$.

The *complement* of a set A is the set A' of everything that is not an element of A. This makes sense only in the context of some "universal set" U containing all the elements we are discussing.

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

Here the symbol $\notin$ means "is not an element of." If U is the set of integers, for example, then $\{1, 2\}'$ is the set of integers other than 1 or 2. The set $\{1, 2\}'$ would be different if U were the set of all real numbers or some other universe.

Two other important operations involving sets are *union* and *intersection*. The union of A and B (sometimes referred to as "A union B") is the set

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

and the intersection of A and B is

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

For example,

$$\{1, 2, 3, 4\} \cup \{2, 4, 6, 8\} = \{1, 2, 3, 4, 6, 8\}$$
$$\{1, 2, 3, 4\} \cap \{2, 4, 6, 8\} = \{2, 4\}$$