Computer Science
and Applied Mathematics

COMPUTABILITY, COMPLEXITY, AND LANGUAGES

FUNDAMENTALS OF THEORETICAL COMPUTER SCIENCE

Martin D. Davis and Elaine J. Weyuker

Computability, Complexity, and Languages

Fundamentals of Theoretical Computer Science

Martin D. Davis
Elaine J. Weyuker

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
New York, New York

1983

ACADEMIC PRESS
A Subsidiary of Harcourt Brace Jovanovi
New York London
Paris San Diego San Franc

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
24/28 Oval Road, London NW1 7DX

Library of Congress Cataloging in Publication Data

Davis, Martin, Date
Computability, complexity, and languages

(Computer science and applied mathematics)
Includes index.
1. Machine theory 2 Computational complexity
3. Formal languages I. Weyuker, Elaine J
II. Title. III. Series
QA267.D38 1983 001 64'01 83-2727
ISBN 0-12-206380-5

PRINTED IN THE UNITED STATES OF AMERICA

83 84 85 86 9 8 7 6 5 4 3 2 1

Preface

Theoretical computer science is the mathematical study of models of computation. As such, it originated in the 1930s, well before the existence of modern computers, in the work of the logicians Church, Gödel, Kleene, Post, and Turing. This early work has had a profound influence on the practical and theoretical development of computer science. Not only has the Turing-machine model proved basic for theory, but the work of these pioneers presaged many aspects of computational practice that are now commonplace and whose intellectual antecedents are typically unknown to users. Included among these are the existence in principle of all-purpose (or universal) digital computers, the concept of a program as a list of instructions in a formal language, the possibility of interpretive programs, the duality between software and hardware, and the representation of languages by formal structures based on productions. While the spotlight in computer science has tended to fall on the truly breathtaking technological advances that have been taking place, important work in the foundations of the subject has continued as well. It is our purpose in writing this book to provide an introduction to the various aspects of theoretical computer science for undergraduate and graduate students that is sufficiently comprehensive that the professional literature of treatises and research papers will become accessible to our readers.

We are dealing with a very young field that is still finding itself. Computer scientists have by no means been unanimous in judging which parts of the subject will turn out to have enduring significance. In this situation, fraught with peril for authors, we have attempted to select topics that have already achieved a polished classic form, and that we believe will play an important role in future research.

We have assumed that many of our readers will have had little experience with mathematical proof, but that almost all of them have had substantial programming experience. Thus the first chapter contains an introduction to the use of proofs in mathematics in addition to the usual explanation of terminology and notation. We then proceed to take advantage of the reader's background by developing computability theory in the context of an extremely simple abstract programming language. By systematic use of a macro expansion technique, the surprising power of the language is demonstrated. This culminates in a universal program, which is written in all detail on a single page. By a series of simulations, we then obtain the equivalence of various different formulations of computability, including Turing's. Our point of view with respect to these simulations is that it should not be the reader's responsibility, at this stage, to fill in the details of vaguely sketched arguments, but rather that it is our responsibility as authors to arrange matters so that the simulations can be exhibited simply, clearly, and completely.

This material, in various preliminary forms, has been used with undergraduate and graduate students at New York University, Brooklyn College, The Scuola Matematica Interuniversitaria—Perugia, The University of California—Berkeley, The University of California—Santa Barbara, and Worcester Polytechnic Institute.

Although it has been our practice to cover the material from the second part of the book on formal languages after the first part, the chapters on regular and on context-free languages can be read immediately after Chapter 1. The Chomsky–Schützenberger representation theorem for context-free languages is used to develop their relation to pushdown automata in a way that we believe is clarifying. Part 3 is an exposition of the aspects of logic that we think are important for computer science and can also be read immediately following Chapter 1. Each of the three chapters of Part 4 introduces an important theory of computational complexity, concluding with the theory of **NP**-completeness. Part 5 contains an introduction to advanced recursion theory, includes a number of topics that have had fruitful analogs in the theory of polynomial-time computability, and concludes with an introduction to priority constructions for recursively enumerable Turing degrees. The anomalies revealed by these constructions must be taken into account in efforts to understand the underlying nature of algorithms, even though there is no reason to believe that the specific algorithms generated will prove useful in practice.

Because many of the chapters are independent of one another, this book can be used in various ways. There is more than enough material for a full-year course at the graduate level on *theory of computation*. We have used the unstarred sections of Chapters 1–6 and Chapter 8 in a successful one-semester junior-level course, Introduction to Theory of Computation, at New York University. A course on *finite automata and formal* *languages* could be based on Chapters 1, 8, and 9. A semester or quarter course on *logic for computer scientists* could be based on selections from Parts 1 and 3. Many other arrangements and courses are possible, as should be apparent from the dependency graph, which follows. It is our hope, however, that this book will help readers to see theoretical computer science not as a fragmented list of discrete topics, but rather as a unified subject drawing on powerful mathematical methods and on intuitions derived from experience with computing technology to give valuable insights into a vital new area of human knowledge.

Note to the Reader

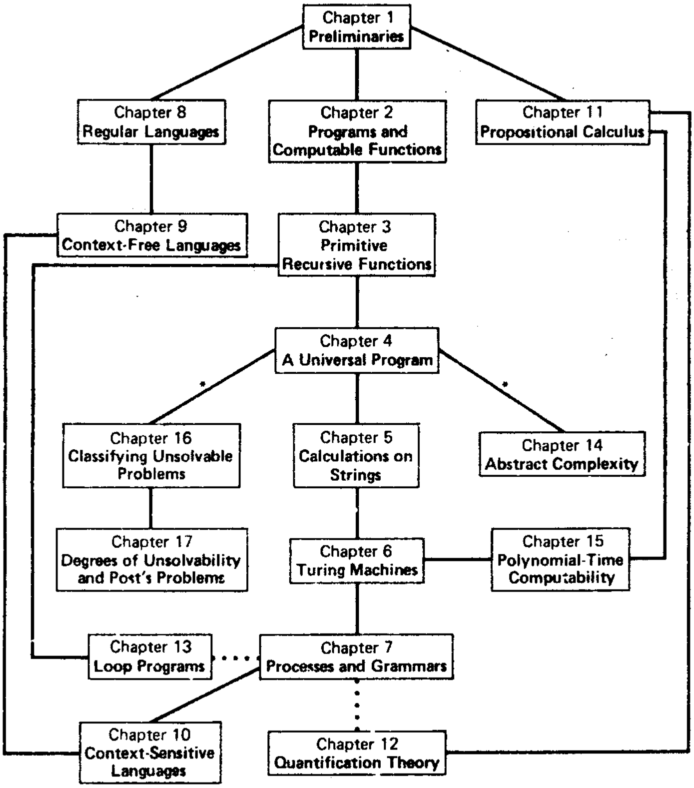
Many readers will wish to begin with Chapter 2, using the material of Chapter 1 for reference as required. Readers who enjoy skipping around will find the *dependency graph* useful.

A reference to Theorem 8.1 is to Theorem 8.1 of the chapter in which the reference is made. When a reference is to a theorem in another chapter, the chapter is specified. The same system is used in referring to numbered formulas and to exercises.

Acknowledgments

It is a pleasure to acknowledge the help we have received. Charlene Herring, Debbie Herring, Barry Jacobs, and Joseph Miller made their student classroom notes available to us. James Cox, Keith Harrow, Steve Henkind, Karen Lemone, Colm O'Dunlaing, and James Robinett provided helpful comments and corrections. Stewart Weiss was kind enough to redraw one of the figures. Thomas Ostrand, Norman Shulman, Louis Salkind, Ron Sigal, Patricia Teller, and Elia Weixelbaum were particularly generous with their time, and devoted many hours to helping us. We are especially grateful to them.

Dependency Graph



A solid line between two chapters indicates the dependence of the unstarred sections of the higher numbered chapter on the unstarred sections of the lower numbered chapter. An asterisk next to a solid line indicates that knowledge of the starred sections of the lower numbered chapter is also assumed. A dotted line shows that knowledge of the unstarred sections of the lower numbered chapter is assumed for the starred sections of the higher numbered chapter.

Contents

Part 2 GRAMMARS AND AUTOMATA

Chapter 8 Regular Languages

Chapter 9 Context-Free Languages

Chapter 10 Context-Sensitive Languages

Part 3 LOGIC

Chapter 11 Propositional Calculus

Chapter 12 Quantification Theory

Part 4 COMPLEXITY

Chapter 13 Loop Programs

Chapter 14 Abstract Complexity

Chapter 15 Polynomial-Time Computability

Part 5 UNSOLVABILITY

Chapter 16 Classifying Unsolvable Problems

Chapter 17 Degrees of Unsolvability and Post's Problem

CHAPTER 1

Preliminaries

1. Sets and n-tuples

We shall often be dealing with *sets* of objects of some definite kind. Thinking of a collection of entities as a *set* simply amounts to a decision to regard the whole collection as a single object. We shall use the word *class* as synonymous with *set*. In particular we write N for the set of *natural numbers* 0, 1, 2, 3, ... In this book the word *number* will always mean *natural number* except in contexts where the contrary is explicitly stated.

We write

$$a \in S$$

to mean that a belongs to S or, equivalently, is a member of the set S, and

$$a \notin S$$

to mean that a does not belong to S. It is useful to speak of the *empty set*, written $\varnothing$, which has no members. The equation $R = S$, where R and S are sets, means that R and S are *identical as sets*, that is, that they have exactly the same members. We write $R \subseteq S$ and speak of R as a *subset* of S to mean that every element of R is also an element of S. Thus, $R = S$ if and only if $R \subseteq S$ and $S \subseteq R$. Note also that for any set R, $\varnothing \subseteq R$ and $R \subseteq R$. We write $R \subset S$ to indicate that $R \subseteq S$ but $R \neq S$. In this case R is called a *proper subset* of S. If R and S are sets, we write $R \cup S$ for the *union* of R and S, that is the collection of all objects which are members of either R or S or both. $R \cap S$, the *intersection* of R and S, is the set of all objects which belong to both R and S. $R - S$, the set of all objects which belong to R and do not belong to S, is the *difference* between R and S. S may contain objects not in R. Thus $R - S = R - (R \cap S)$. Often we will be working in contexts where all sets being considered are subsets of some fixed set D (sometimes called a *domain* or a *universe*). In such a case we write $\bar{S}$ for $D - S$, and call

$\bar{S}$ the *complement* of S. Most frequently we shall be writing $\bar{S}$ for $N - S$. The De Morgan identities

$$\overline{R \cup S} = \bar{R} \cap \bar{S},$$

$$\overline{R \cap S} = \bar{R} \cup \bar{S}$$

are very useful; they are easy to check and any reader not already familiar with them should do so. We write

$$\{a_1, a_2, \ldots, a_n\}$$

for the set consisting of the n objects $a_1, a_2, \ldots, a_n$. Sets which can be written in this form as well as the empty set are called *finite*. Sets which are not finite, e.g., N, are called *infinite*. It should be carefully noted that a and $\{a\}$ are not the same thing. In particular, $a \in S$ is true if and only if $\{a\} \subseteq S$. Since two sets are equal if and only if they have the same members, it follows that, for example, $\{a, b, c\} = \{a, c, b\} = \{b, a, c\}$. That is, the order in which we may choose to write the members of a set is irrelevant. Where order is important, we speak instead of an n-tuple or a *list*. We write n-tuples using parentheses rather than curly braces:

$$(a_1, \ldots, a_n).$$

Naturally, the elements making up an n-tuple need not be distinct. Thus (4, 1, 4, 2) is a 4-tuple. A 2-tuple is called an *ordered pair* and a 3-tuple is called an *ordered triple*. Unlike the case for sets of one object, we *do not distinguish between the object a and the 1-tuple* (a). The crucial property of n-tuples is

$$(a_1, a_2, \ldots, a_n) = (b_1, b_2, \ldots, b_n)$$

if and only if

$$a_1 = b_1, \quad a_2 = b_2, \quad \ldots, \quad \text{and} \quad a_n = b_n.$$

If $S_1, S_2, \ldots, S_n$ are given sets, then we write $S_1 \times S_2 \times \cdots \times S_n$ for the set of all n-tuples $(a_1, a_2, \ldots, a_n)$ such that $a_1 \in S_1, a_2 \in S_2, \ldots, a_n \in S_n$. $S_1 \times S_2 \times \cdots \times S_n$ is sometimes called the Cartesian product of $S_1, S_2, \ldots, S_n$. In case $S_1 = S_2 = \cdots = S_n = S$ we write S^n for the Cartesian product $S_1 \times S_2 \times \cdots \times S_n$.

2. Functions

Functions play an important role in virtually every branch of pure and applied mathematics. We may define a function simply as a set f, all of whose members are ordered pairs and which has the special property

$$(a, b) \in f \text{ and } (a, c) \in f \quad \text{implies} \quad b = c.$$

However, intuitively it is more helpful to think of the pairs listed as the rows of a table. For f a function, one writes $f(a) = b$ to mean that $(a, b) \in f$; the definition of function ensures that for each a there can be at most one such b. The set of all a such that $(a, b) \in f$ for some b is called the *domain* of f. The set of all $f(a)$ for a in the domain of f is called the *range* of f.

As an example, let f be the set of ordered pairs (n, n^2) for $n \in N$. Then, for each $n \in N$, $f(n) = n^2$. The domain of f is N. The range of f is the set of perfect squares.

Functions f are often specified by *algorithms* which provide procedures for obtaining $f(a)$ from a. This method of specifying functions is particularly important in computer science. However, as we shall see in Chapter 4, it is quite possible to possess an algorithm which specifies a function without being able to tell which elements belong to its domain. This makes the notion of a so-called *partial function* play a central role in computability theory. A *partial function on a set S* is simply a function whose domain is a subset of S. An example of a partial function on N is given by $g(n) = \sqrt{n}$, where the domain of g is the set of perfect squares. If f is a partial function on S and $a \in S$, then we write $f(a)\downarrow$ and say that $f(a)$ is *defined* to indicate that a is in the domain of f; if a is not in the domain of f, we write $f(a)\uparrow$ and say that $f(a)$ is *undefined*. If a partial function on S has the domain S, then it is called *total*. Finally, we should mention that the empty set $\varnothing$ is itself a function. Considered as a partial function on some set S, *it is nowhere defined*.

For a partial function f on a Cartesian product $S_1 \times S_2 \times \cdots \times S_n$, we write $f(a_1, \ldots, a_n)$ rather than $f((a_1, \ldots, a_n))$. A partial function f on a set S^n is called an *n-ary partial function on S*, or a function of n variables on S. We use *unary* and *binary* for 1-ary and 2-ary, respectively. For n-ary partial functions, we often write $f(x_1, \ldots, x_n)$ instead of f as a way of showing explicitly that f is n-ary.

3. Alphabets and Strings

An *alphabet* is simply some finite nonempty set A of objects called *symbols*. An n-tuple of symbols of A is called a *word* or a *string* on A. Instead of writing a word as $(a_1, a_2, \ldots, a_n)$ we write simply $a_1 a_2 \cdots a_n$. If $u = a_1 a_2 \cdots a_n$, then we say that n is the length of u and write $|u| = n$. We allow a unique null word, written 0, of length 0. (The reason for using the same symbol for the number zero and the null word will become clear in Chapter 5.)The set of all words on the alphabet A is written A^*. Any subset of A^* is called a *language on A* or a *language with alphabet A*. We do *not* distinguish between a symbol $a \in A$ and the word of length 1 consisting of that symbol.

If $u, v \in A^*$, then we write $\widehat{uv}$ for the word obtained by placing the string v after the string u. For example, if $A = \{a, b, c\}$, $u = bab$, and $v = caa$, then

$$\widehat{uv} = babcaa \qquad \text{and} \qquad \widehat{vu} = caabab.$$

Where no confusion can result, we write uv instead of $\widehat{uv}$. It is obvious that for all u,

$$u0 = 0u = u,$$

and that for all u, v, w,

$$u(vw) = (uv)w.$$

Also, if either $uv = uw$ or $vu = wu$, then $v = w$.

If u is a string, and $n \in N$, $n > 0$, we write

$$u^{[n]} = \underbrace{u\,u \cdots u}_{n}.$$

We also write $u^{[0]} = 0$. We use the square brackets to avoid confusion with numerical exponentiation.

If $u \in A^*$, we write u^R for u written backward; i.e., if $u = a_1 a_2 \cdots a_n$, for $a_1, \ldots, a_n \in A$, then $u^R = a_n \cdots a_2 a_1$. Clearly, $0^R = 0$ and $(uv)^R = v^R u^R$ for $u, v \in A^*$.

4. Predicates

By a *predicate* or a *Boolean-valued function* on a set S we mean a *total* function P on S such that for each $a \in S$, either

$$P(a) = \text{TRUE} \qquad \text{or} \qquad P(a) = \text{FALSE},$$

where TRUE and FALSE are a pair of distinct objects called *truth values*. We often say *$P(a)$ is true* for $P(a) = $ TRUE, and *$P(a)$ is false* for $P(a) = $ FALSE. For our purposes it is useful to identify the truth values with specific numbers, so we set

$$\text{TRUE} = 1 \qquad \text{and} \qquad \text{FALSE} = 0.$$

Thus, a predicate is a special kind of function with values in N. Predicates on a set S are usually specified by expressions which become statements, either true or false, when variables in the expression are replaced by symbols designating fixed elements of S. Thus the expression

$$x < 5$$