Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York, NY 10022, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia



First published 1985

Printed in Great Britain

*Library of Congress Cataloging in Publication Data*
Salomaa, Arto.
Computation and automata.
(Encyclopedia of mathematics and its applications; v. 25)
Bibliography: p.
Includes index.
1. Computable functions. 2. Computational complexity. 3. Sequential machine theory. I. Title.
II. Series.
QA9.59.S25 1985 511 84-17571
ISBN 0 521 30245 5

# Editor's Statement

A large body of mathematics consists of facts that can be presented and described much like any other natural phenomenon. These facts, at times explicitly brought out as theorems, at other times concealed within a proof, make up most of the applications of mathematics, and are the most likely to survive change of style and of interest.

This ENCYCLOPEDIA will attempt to present the factual body of all mathematics. Clarity of exposition, accessibility to the non-specialist, and a thorough bibliography are required of each author. Volumes will appear in no particular order, but will be organized into sections, each one comprising a recognizable branch of present-day mathematics. Numbers of volumes and sections will be reconsidered as times and needs change.

It is hoped that this enterprise will make mathematics more widely used where it is needed, and more accessible in fields in which it can be applied but where it has not yet penetrated because of insufficient information.

GIAN-CARLO ROTA

# Foreword

The last twenty years have witnessed most vigorous growth in areas of mathematical study connected with computers and computer science. The enormous development of computers and the resulting profound changes in scientific methodology have opened new horizons for the science of mathematics at a speed without parallel during the long history of mathematics.

The following two observations should be kept in mind when reading the present monograph. First, various developments in mathematics have directly initiated the "beginning" of computers and computer science. Second, advances in computer science have induced very vigorous developments in certain branches of mathematics. More specifically, the second of these observations refers to the growing importance of discrete mathematics—and we are now witnessing only the very beginning of the influence of discrete mathematics.

Because of reasons outlined above, mathematics plays a central role in the foundations of computer science. A number of significant research areas can be listed in this connection. It is interesting to notice that these areas also reflect the historical development of computer science.

1. The classical *computability theory* initiated by the work of Gödel, Tarski, Church, Post, Turing, and Kleene occupies a central role. This area is rooted in mathematical logic.

2. In the classical *formal language and automata theory* the central notions are those of an automaton, a grammar, and a language. Apart from

developments in area (1), the work of Chomsky on the foundations of natural languages, as well as the work of Post concerning rewriting systems, should be mentioned here. It is, however, fascinating to observe that the modern theory of formal languages and rewriting systems was initiated by the work of the Norwegian mathematician Axel Thue at the beginning of this century!

3. An area initiated in the sixties is *complexity theory*. The performance of an algorithm is investigated. The central notions are those of a tractable and an intractable problem. This area is gaining in importance because of several reasons, one of them being the advances in area (4).

4. Quite recent developments concerning the security of computer systems have increased the importance of *cryptography* to a great extent. Moreover, the idea of public key cryptography is of specific theoretical interest and has drastically changed our ideas concerning what is doable in communication systems.

Areas (1) through (4) constitute the core of the present monograph. Many other important areas dealing with the mathematical foundations of computer science (e.g., semantics and the theory of correctness of programming languages, the theory of data structures, and the theory of data bases) lie beyond the scope of the present monograph and will, hopefully, be presented in other books in this series.

All the areas listed above comprise a fascinating part of contemporary mathematics that is very dynamic in character, full of challenging problems requiring most interesting and ingenious mathematical techniques.

This monograph provides a very good basis for the understanding of these developments. It presents this fascinating modern area of mathematics in a broad and clear perspective. Because everything is developed essentially from the beginning, even an uninitiated reader can use the monograph as an entry to this area. In spite of this, a glimpse of a number of very recent developments is given.

Grzegorz Rozenberg

# Acknowledgments

It is difficult to list all persons who have in some way or other contributed to this book. Parts of the manuscript were used as lecture notes for courses given at the universities of Turku and Waterloo. I want to thank the participants in these courses, in particular, Juha Honkala and Sheng Yu. Tero Harju, Juha Honkala, Werner Kuich, Valtteri Niemi, and Grzegorz Rozenberg have read through at least some parts of the manuscript and given very useful comments. Moreover, I have benefited from discussions with or comments from Karel Culik II, Jozef Gruska, Helmut Jürgensen, Juhani Karhumäki, Matti Linna, Hermann Maurer, Martti Penttonen, Keijo Ruohonen, Adi Shamir, Emo Welzl, and Derick Wood. The difficult task of typing the manuscript was performed in an excellent fashion by Elisa Mikkola. I want to thank the publisher for excellent and timely editorial work with both the typescript and proofs. Last but not least, I want to acknowledge the continuing support of my wife, children, and other members of the family. In particular, discussions with Ilokivi and Turzan were always very encouraging, and the whole book would not have been possible without Ketta and Korak.

Arto Salomaa

# Contents

CHAPTER 1

# Introduction: Models of Computation

The basic question in the theory of computing can be formulated in any of the following ways: What is computable? For which problems can we construct effective mechanical procedures that solve every instance of the problem? Which problems possess algorithms for their solutions?

Fundamental developments in mathematical logic during the 1930s showed the existence of *unsolvable* problems: No algorithm can possibly exist for the solution of the problem. Thus, the existence of such an algorithm is a logical impossibility—its nonexistence has nothing to do with our ignorance. This state of affairs led to the present formulation of the basic question in the theory of computing. Previously, people always tried to construct an algorithm for every precisely formulated problem until (if ever) the correct algorithm was found. The basic question is of definite practical significance: One should not try to construct algorithms for an unsolvable problem. (There are some notorious examples of such attempts in the past.)

A *model of computation* is necessary for establishing unsolvability. If one wants to show that no algorithm for a specific problem exists, one must have a precise definition of an algorithm. The situation is different in establishing solvability: It suffices to exhibit some particular procedure that is effective in the intuitive sense. (We use the terms *algorithm* and *effective procedure* synonymously. There are some obvious requirements every intui-

tively effective procedure has to satisfy. At the moment we do not try to list such requirements.)

We are now confronted with the necessity of formalizing a notion of a model of computation that is general enough to cover all conceivable computers, as well as our intuitive notion of an algorithm. Some initial observations are in order.

Let us assume that the algorithms we want to formalize compute functions mapping the set of nonnegative integers into the same set. Although this is not important at this point, we could observe that our assumption is no essential restriction of generality. This is due to the fact that other input and output formats can be encoded into nonnegative integers.

After having somehow defined our general model of computation, denoted by $MC$, we observe that each specific instance of the model possesses a finitary description; that is, it can be described in terms of a formula or finitely many words. By enumerating these descriptions, we obtain an enumeration $MC_1, MC_2, \ldots$ of all specific instances of our general model of computation. In this enumeration, each $MC_i$ represents some particular algorithm for computing a function from nonnegative integers into nonnegative integers. Denote by $MC_i(j)$ the value of the function computed by $MC_i$ for the argument value $j$.

Define a function $f(x)$ by

$$f(x) = MC_x(x) + 1. \tag{1}$$

Clearly, the following is an algorithm (in the intuitive sense) to compute the function $f(x)$. Given an input $x$, start the algorithm $MC_x$ with the input $x$ and add one to the output.

However, is there any specific algorithm among our formalized $MC$-models that would compute the function $f(x)$? The answer is no, and the argument is an indirect one. Assume that $MC_t$ would give rise to such an algorithm, where $t$ is some natural number. Hence, for all $x$,

$$f(x) = MC_t(x). \tag{2}$$

A contradiction now arises by substituting the value $t$ for the variable $x$ in both (1) and (2).

This contradiction, referred to as the *dilemma of diagonalization,* shows that independently of our model of computation—indeed, we did not specify the $MC$-model in any way—there will be algorithms not formalized by the model.

There is a simple and natural way to avoid the dilemma of diagonalization. We have assumed so far that the $MC_i$-algorithms are defined everywhere: For all input $j$, the algorithm $MC_i$ produces an output. This assumption is unreasonable from many points of view, one of which is computer programming; we cannot be sure that every program produces an output for every input. Therefore, we should allow also the possibility that some of the

$MC_i$-algorithms enter an infinite loop for some inputs $j$ and, consequently, do not produce any output for such a $j$. Moreover, the set of such values $j$ is not known a priori.

Thus, some algorithms in the list

$$MC_1, MC_2, \ldots \tag{3}$$

produce an output only for some of the possible inputs; that is, the corresponding functions are not defined for all nonnegative integers. The dilemma of diagonalization does not arise after the inclusion of such *partial* functions among the functions computed by the algorithms of (3). Indeed, the argument presented above does not lead to a contradiction because $MC_t(t)$ is not necessarily defined.

The general model of computation, now referred to as a *Turing machine,* was introduced quite a long time before the advent of electronic computers. Turing machines constitute by far the most widely used general model of computation. Other general models discussed later in this book are *Markov algorithms, Post systems, grammars,* and *L systems.* Each of these models leads to a list such as (3), where partial functions are also included. All models are also equivalent in the sense that they define the same set of solvable problems or computable functions. This is understood in a sense made precise later; also, the input and output formats are taken into account. For instance, grammars naturally define languages and, consequently, an input-output format associated to computing function values is rather unsuitable for grammars.

We have considered only the general question of characterizing the class of solvable problems. This question was referred to as basic in the theory of computing. It led to a discussion of general models of computation.

More specific questions in the theory of computing deal with the *complexity* of solvable problems. Is a problem $P_1$ more difficult that $P_2$ in the sense that every algorithm for $P_1$ is more complex (for instance, in terms of time or memory space needed) than a reasonable algorithm for $P_2$? What is a reasonable classification of problems in terms of complexity? Which problems are so complex that they can be classified as *intractable* in the sense that all conceivable computers require an unmanageable amount of time for solving the problem?

Undoubtedly, such questions are of crucial importance from the point of view of practical computing. A problem is not yet settled if it is known to be solvable or computable and remains intractable at the same time. As a typical example, many recent results in cryptography are based on the assumption that the factorization of the product of two large primes is impossible in practice. More specifically, if we know a large number $n$ consisting of, for example, 200 digits and if we also know that $n$ is the product of two large primes, it is still impossible for us to find the two primes. This assumption is reasonable because the problem described is intractable,

at least in view of the factoring algorithms known at present. Of course, from a merely theoretical point of view where complexity is not considered, such a factoring algorithm can be trivially constructed.

Such specific questions lead to more specific models of computing. The latter are obtained either by imposing certain restrictions on Turing machines or else by some direct construction. Also, such specific models will be discussed in the sequel. Of particular importance is the *finite automaton*. It is a model of a strictly finitary computing device: The automaton is not capable of increasing any of its resources during the computation.

It is clear that no model of computation is suitable for all situations; modifications and even entirely new models are needed to match new developments. Theoretical computer science by now has a history long enough to justify a discussion about good and bad models. The theory is mature enough to produce a great variety of different models of computation and prove some interesting properties concerning them. Good models should be general enough so that they are not too closely linked with any particular situation or problem in computing—they should be able to lead the way. On the other hand, they should not be too abstract. Restrictions on a good model should converge, step by step, to some area of real practical significance. A typical example is some restrictions of abstract grammars especially suitable for considerations concerning parsing. The resulting aspects of parsing are essential in compiler construction.

To summarize: A good model represents a well-balanced abstraction of a real practical situation—not too far from and not too close to the real thing.

*Formal languages* constitute a descriptive tool for models of computation, both in regard to the input-output format and the mode of operation. Formal language theory is by its very essence an interdisciplinary area of science; the need for a formal grammatical description arises in various scientific disciplines, ranging from linguistics to biology. Therefore, appropriate aspects of formal language theory will be of crucial importance in this book.

CHAPTER 2

# Rudiments of Language Theory

## 2.1. LANGUAGES AND REWRITING SYSTEMS

Both natural and programming languages can be viewed as sets of sentences—that is, finite strings of elements of some basic vocabulary. The notion of a language introduced in this section is very general. It certainly includes both natural and programming languages and also all kinds of nonsense languages one might think of. Traditionally, formal language theory is concerned with the syntactic specification of a language rather than with any semantic issues. A syntactic specification of a language with finitely many sentences can be given, at least in principle, by listing the sentences. This is not possible for languages with infinitely many sentences. The main task of formal language theory is the study of finitary specifications of infinite languages.

The basic theory of computation, as well as of its various branches, such as cryptography, is inseparably connected with language theory. The input and output sets of a computational device can be viewed as languages, and—more profoundly—models of computation can be identified with classes of language specifications, in a sense to be made more precise. Thus, for instance, Turing machines can be identified with phrase-structure grammars and finite automata with regular grammars.

We begin by introducing some notions and terminology fundamental to all our discussions.

An **alphabet** is a finite, nonempty set. The elements of an alphabet, which we might call $\Sigma$, are referred to as **letters,** or **symbols.** A **word** over an alphabet $\Sigma$ is a finite string consisting of zero or more letters of $\Sigma$, in which the same letter may occur several times. The string consisting of zero letters is called the **empty word,** written $\lambda$. For instance, $\lambda$, 0, 10, 1011, and 00000 are words over the alphabet $\Sigma = \{0, 1\}$. The set of all words (resp. all nonempty words) over an alphabet $\Sigma$ is denoted by $\Sigma^*$ (resp. $\Sigma^+$). The sets $\Sigma^*$ and $\Sigma^+$ are infinite for any $\Sigma$. Algebraically speaking, $\Sigma^*$ and $\Sigma^+$ are the free monoid (with the identity $\lambda$) and the free semigroup generated by $\Sigma$.

The reader should keep in mind that the basic set $\Sigma$, its elements, and strings of its elements could equally well be called a *vocabulary, words,* and *sentences,* respectively. This would reflect an approach with applications mainly in the area of natural languages. In this book, we use the standard mathematical terminology introduced above.

For words $w_1$ and $w_2$, the juxtaposition $w_1w_2$ is called the **catenation** (or concatenation) of $w_1$ and $w_2$. The empty word is an identity with respect to catenation: $w\lambda = \lambda w = w$ holds for all words $w$. Because catenation is associative, the notation $w^i$, where $i$ is a positive integer, is used in the customary sense. By definition, $w^0$ is the empty word, $\lambda$.

The **length** of a word $w$, denoted by $|w|$, is the number of letters in $w$ when each letter is counted as many times as it occurs. Again by definition, $|\lambda| = 0$. The length function possesses some of the formal properties of logarithm:

$$|w_1w_2| = |w_1| + |w_2|, \qquad |w^i| = i|w|$$

for all words $w$ and integers $i \geq 0$.

A word $w$ is a **subword** (or a *factor*) of a word $u$ if there are words $x$ and $y$ such that $u = xwy$. Furthermore, if $x = \lambda$ (resp. $y = \lambda$), then $w$ is called an **initial** subword, or a **prefix,** of $u$ (resp. a **final** subword or a **suffix** of $u$).

Subsets of $\Sigma^*$ are referred to as **formal languages**—or, briefly, **languages**—over $\Sigma$.

Thus, this definition is very general: A formal language need not have any form whatsoever! The reader might also find our terminology somewhat unusual in general. A language should consist of sentences rather than of words, as is the case in our terminology. However, as already pointed out above, this is irrelevant and depends merely on the choice of the basic terminology; we have chosen the "neutral" mathematical terminology.

For instance,

$$L_1 = \{\lambda, 0, 010, 1110\} \quad \text{and} \quad L_2 = \{0^p \mid p \text{ prime}\}$$

are languages over the alphabet $\Sigma = \{0, 1\}$, the former being finite and the latter infinite. Here, $L_2$ is also a language over the alphabet $\Sigma_1 = \{0\}$. In general, if $L$ is a language over the alphabet $\Sigma_1$ and $\Sigma$ is an alphabet containing $\Sigma_1$, then $L$ is also a language over $\Sigma$. However, when we speak of the *alphabet of a language* $L$, denoted by ALPH($L$), then we mean the smallest

alphabet $\Sigma$ such that $L$ is a language over $\Sigma$. Thus, $\mathrm{ALPH}(L_1) = \{0, 1\}$ and $\mathrm{ALPH}(L_2) = \{0\}$. If $L$ consists of a single word, $L = \{w\}$, then we write simply $\mathrm{ALPH}(w)$ instead of $\mathrm{ALPH}(\{w\})$. In general, we identify elements $x$ and singleton sets $\{x\}$ whenever there is no danger of confusion.

Specific families of languages are often conveniently characterized in terms of operations defined for languages: The family consists of all languages obtainable from certain given languages by certain operations. We now define some of the most-common operations. Others will be defined later on.

Regarding languages as sets, we may immediately define the **Boolean operations** of union, intersection, complementation, and difference in the natural fashion. The customary notations $L \cup L'$, $L \cap L'$, $\sim L$ and $L - L'$ are used. In defining the complement of $L$, $\sim L$, we often consider $\mathrm{ALPH}(L) = \Sigma$ : $\sim L$ consists of all words in $\Sigma^*$ that are not in $L$. Thus,

$$\sim L = \Sigma^* - L.$$

(This is done in order to avoid any ambiguity in the definition of complementation. When defining the other Boolean operations, the alphabet need not be considered. One should, however, be careful; if complement is defined using ALPH, then some of the customary formulas are not necessarily valid. An example of such a formula is $\sim\sim L = L$.)

The **catenation** (or *product*) **of two languages** $L_1$ and $L_2$ is defined by

$$L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \quad \text{and} \quad w_2 \in L_2\}.$$

The notation $L^i$ is extended to apply to the catenation of languages. By definition, $L^0 = (\lambda)$. Observe that this definition guarantees that the customary equations

$$L^i L^j = L^{i+j} \quad \text{and} \quad (L^i)^j = L^{ij}$$

hold for all languages $L$ and nonnegative integers $i$ and $j$. Observe also that the empty language, $\varnothing$, is not the same as the language $\{\lambda\}$. Indeed, $\varnothing$ and $\{\lambda\}$ can be considered as zero and unit elements with respect to catenation because, for any language $L$,

$$L\varnothing = \varnothing L = \varnothing, \qquad L\{\lambda\} = \{\lambda\}L = L.$$

The **catenation closure** of a language $L$, $L^*$, is defined to be the union of all powers of $L$:

$$L^* = \bigcup_{i=0}^{\infty} L^i.$$

The **$\lambda$-free catenation closure** of $L$, $L^+$, is defined to be the union of all positive powers of $L$:

$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

Thus, a word is in $L^+$ iff it is obtained by catenating a finite number of words belonging to $L$. The empty word, $\lambda$, is in $L^*$ for every $L$ (including $L = \emptyset$) because $L^0 = \{\lambda\}$. Observe also that the notations $\Sigma^*$ and $\Sigma^+$ introduced previously are in accordance with the definition of the operations $L^*$ and $L^+$ if $\Sigma$ is viewed as the finite language consisting of all single-letter words. For instance,

$$\{a^{2n} \mid n \geq 1\} = \{a^2\}^+ \quad \text{and} \quad \{a^{7n+3} \mid n \geq 0\} = \{a^7\}^*\{a^3\}.$$

An operation of crucial importance in language theory is the operation of morphism. A mapping $h : \Sigma^* \to \Delta^*$, where $\Sigma$ and $\Delta$ are alphabets, satisfying the condition

$$h(ww') = h(w)h(w'), \qquad \text{for all words } w \text{ and } w' \tag{1}$$

is called a **morphism.** For languages $L$ over $\Sigma$, we define

$$h(L) = \{h(w) \mid w \text{ is in L}\}.$$

(Again, algebraically speaking, a morphism of languages is a monoid morphism linearly extended to subsets of monoids.) In view of the condition in (1), to define a morphism $h$, it suffices to list all the words $h(a)$, where $a$ ranges over all the finitely many letters of $\Sigma$. A morphism $h$ is called **nonerasing** (resp. **letter-to-letter**) if $h(a) \neq \lambda$ (resp. $h(a)$ is a letter) for every $a$ in $\Sigma$.

We have pointed out that a finite language can be defined, at least in principle, by listing all the words in it, whereas such a definition is not possible for infinite languages. We have already seen how to define infinite languages by specifying a property that must be satisfied by the words in the language. An example is the language $\{0^p \mid p \text{ prime}\}$. The operations introduced above give a way of defining infinite languages because each of the operations $\sim$, $*$, and $^+$ yields an infinite language when applied to a finite language containing at least one nonempty word. For instance, we may consider all languages obtainable from the *atomic* languages $\emptyset$ and $\{a\}$, where $a$ ranges over the letters of some alphabet $\Sigma$, by finitely many applications of the operations introduced above. Such languages are called *regular* in Chapter 3, where it will be also seen that we need only a few of the operations introduced above to get all these languages.

We shall introduce a general model for the definition of languages by means of "legal" derivations. The model is referred to as a **rewriting system.** The notion of a (phrase-structure) **grammar** is obtainable from this model by providing it with an input and output format. Before introducing this model, we still want to consider four examples of a somewhat more sophisticated nature than the examples mentioned above. The first three examples deal with operations and are also of general theoretical interest: Example 2.1 in regard to operations in general, Example 2.2 for regular languages, and Example 2.3 for cryptography. The fourth example introduces the notion of a rewriting system.

**Example 2.1.** Consider the language $L$ over the alphabet $\{a, b, c\}$ consisting of all words of the form

$$c^i w c^j, \qquad i \geq 0, j \geq 0,$$

where $w$ is the empty word, the letter $a$ is a prefix of $w$, or the letter $b$ is a suffix of $w$. (Thus, for instance, $\lambda$, $c^3$, $cacbac^2$, $ca$, and $bc$ are all in $L$, whereas none of the words $ba$, $c^3bca^3c$, $c^2bc^7a$ is in $L$.) Although $L$ misses many words over the alphabet $\{a, b, c\}$, we claim that

$$L^2 = \{a, b, c\}^*. \tag{2}$$

Consequently, since $\lambda$ is in $L$, $L^i = \{a, b, c\}^*$ for every $i \geq 2$.

To establish the claim in (2), we prove that an arbitrary given word $x$ over $\{a, b, c\}$ is in $L^2$. This is obvious if $x = \lambda$. If $a$ is a prefix of $x$, then $x$ is in $L$ and, hence, also in $L^2$. (Observe that $L^2$ contains $L$.) If $b$ is a prefix of $x$, we may write $x$ in the form $x = bz$ or $x = bybz$, for some words $y$ and $z$ such that $b$ is not in ALPH($z$). Clearly, the words $b$, $byb$, and $z$ are in $L$ and, consequently, $x$ is in $L^2$. Finally, let $c$ be a prefix of $x$. If $b$ is not in ALPH($x$), then clearly $x$ is in $L$. Otherwise, we may write $x$ in the form

$$x = c^i ybz,$$

for some $i \geq 0$ and words $y$ and $z$ such that $b$ is not in ALPH($z$). Again, both $c^i yb$ and $z$ are in $L$ and, consequently, $x$ is in $L^2$. Since we have exhausted all cases, the claim in (2) follows. The reader might want to prove (2) by considering the cases: $a$ occurs in $x$ and $a$ does not occur in $x$.

**Example 2.2.** Define the language $L$ by $L = \{ababa\}^*$. Thus, $L$ consists of the empty word $\lambda$ and of all words of the form $(ababa)^n$, where $n \geq 1$. We want to show that $L$ can be obtained from the atomic languages $\emptyset$, $\{a\}$, and $\{b\}$ without using the star operation. (The definition above shows how $L$ is obtained from the atomic languages by the operations of star and catenation.) We claim that $L$ can be obtained from the atomic languages by the operations of catenation, union, and complementation.

Let $\Sigma = \{a, b\}$ and observe that $\sim\emptyset = \Sigma^*$. Observe also that intersection can be expressed in terms of union and complementation:

$$L_1 \cap L_2 = \sim(\sim L_1 \cup \sim L_2)$$

for all languages $L_1$ and $L_2$. Finally, observe that

$$\{\lambda\} = \sim((\{a\} \cup \{b\})\Sigma^*).$$

Consequently, we may use each of the items $\Sigma^*$, $\cap$, and $\{\lambda\}$ without loss of generality in our following considerations.

Since the nonempty words in $L$ are $ababa$, $ababaababa$, $ababaababaababa$, . . . , we conclude that the words

$$ababa, \qquad babaa, \qquad abaab, \qquad baaba, \qquad aabab \tag{3}$$