13-913368-2
Library of Congress Catalogue Card No.: 72-82901

Printed in the United States of America
Current printing (last digit):
10 9 8 7 6 5 4 3 2

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA, PTY. LTD., *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LTD., *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*

For my friends in Czechoslovakia

THEORIES OF ABSTRACT AUTOMATA

THEORIES OF ABSTRACT AUTOMATA

MICHAEL A. ARBIB

Stanford Electronics Laboratories
Stanford University

PRENTICE-HALL, INC.
Englewood Cliffs, N. J.

PREFACE

Abstract automata theory may be defined, approximately, as the mathematical investigation of the *general* questions raised by the study of information-processing systems, be they men or machines. In particular, it focuses on the algebraic and combinatorial problems so raised.

Automata theorists have attacked a wide range of problems. No book could—or should—attempt to summarize all their studies. What this book does do is present an overview of the classical results of automata theory, as well as sample some of the most exciting areas of current research.

This book is perhaps the first advanced textbook in automata theory—its scope is broader than that of any other book on automata theory, and it treats many topics in greater depth than can be found outside specialized research monographs. Yet the book is not exceptionally long, and it is self-contained. A special feature that aids this compactness is that if the idea of a proof can be made clear in a few lines, then the proof will be presented heuristically to aid first-reading comprehension—details can then be supplied by the serious student on a second reading, as he works the clearly marked exercises which are distributed in appropriate places throughout the text.

The reader should have sufficient background in abstract algebra to be able to assimilate a concise treatment of the basic notions of automata theory, or know enough automata theory to be able to use Chapter 2 to acclimate himself to the abstract mathematical approach.

An algebraic background might include the elements of set theory together with a course in group theory, up to and including the Jordan-Hölder Theorem. A reader who is completely at home with such material should have no undue trouble with this book—and need have no previous study of automata theory. This book should be well suited for a first course in automata theory for mathematicians.

A background in automata theory may be provided by any one of the excellent new introductory texts available, such as Booth [1967], Hennie [1968], or Minsky [1967]. I have been asked whether there is need for a text in automata theory intermediate in level between such texts and the present volume. My answer is *no*—but there may be need of intermediate study of abstract algebra. Thus the instructor using this book as text for a second course in automata theory for computer scientists or engineers may need to devote a few lectures to giving exercises and motivation to supplement the summary of mathematical background given in Chapter 2 of this book.

Let us summarize the three parts of this book.

The first part is background. In Chapter 1, we present a mildly mathematical survey of automata theory—an overview of the structures of automata theory and the main questions which we ask about them. This provides a framework wherein we may locate the many topics of Parts II and III. In Chapter 2, we summarize the mathematical concepts needed in abstract automata theory—distinguishing the tools that must be used throughout from those that are needed in one or two special investigations.

The second part might be subtitled "What every literate automata theorist ought to know." It contains my choice of the topics without which one cannot hope to follow the research literature. Chapter 3 treats finite automata, their minimization, circuits which realize them (and the complexity thereof), the languages they characterize, and their relation to semigroups. Chapter 4 treats Turing machines, even polycephalic ones, looks at the functions they compute, studies the sets they define, and introduces the reader to unsolvable decision problems (and proves Gödel's incompleteness theorem as an application). Chapter 5 examines Post's canonical systems, and then concentrates on the context-sensitive and context-free grammars, relating them to push-down automata and linear bounded automata, and proves the basic results on ambiguity and undecidability of context-free languages. By and large, then, Part II contains the material that most automata theorists would want to see included in any thoroughgoing text.

The third part, after a preliminary study of partial recursive functions, is devoted to the presentation of four of the many topics now at the forefront of research in automata theory. They are: complexity of computation, algebraic decomposition theory, stochastic automata, and machines which compute and construct. Here my choice is purely personal—I suspect that the three main omissions which might be lamented are advanced topics in language theory, the theory of algebra automata, and the relation between mathematical logic and automata. But here we begin to leave the domain of the general textbook and start to debate which monographs should best be written to complement the basis which this book seeks to provide.

This book grew from lectures given at Imperial College, in London, in 1964, and at the University of New South Wales, in Sydney, in 1965. It has improved greatly from the comments of my friends and students there, and

in the last three years at Stanford. My main debt is to my fellow automata theorists whose papers, correspondence, and discussion at meetings, have provided much material and stimulation. I have tried to guide the reader to their writings to shed more light on the material I have presented, but have made no attempt to establish priority, or to find references for material which is by now classical (which means, I suppose, that the material had appeared in a textbook by 1963). The book has benefited much from my discussion with Manuel Blum, P. C. Fischer, Shafee Give'on, J. Hartmanis, R. E. Kalman, Bill Kilmer, W. Ogden, J. Rhodes, Fred Roberts, Bill Rounds, P. M. Spira, Don Stanat and Paul Zeiger. To those others whose valued comments are not explicitly acknowledged, my thanks and apologies, and a sincere hope that they will read the book and find that their comments did not fall on deaf ears. Finally, my thanks to Mrs. Rowena Swanson, who monitored the support given by the Air Force Office of Scientific Research, Information Science Directorate, for much of my research reported herein. May this research prove of interest to men of all nations, so that defense yields to the making of friends.

Michael A. Arbib

Stanford, California

A NOTE TO THE READER

All items, except figures, are numbered consecutively within each section. A reference to item **a** means item **a** of the present section; to item **b.a** means item **a** of section **b** of the present chapter; and to item **c.b.a** means item **a** of section **b** of chapter **c** of the present volume.

I have not presumed to tell the reader which sections to omit on a first reading, but I have used a smaller typeface for material which the reader may safely omit on a first reading of the section in which it occurs. Note that this means that if an exercise is set in full-size type, then it is an integral part of the text, and should be read and understood by the reader even if he is in no mood actually to solve the stated problem.

The symbol □ indicates that no more proof will be given of the last numbered proposition that precedes it.

One more comment. This book will be published at least a year after completion of the manuscript. Looking back, I think this book has stood the test of time in that all the material in parts I and II and most of part III, is still essential for the repertoire of the automata theorist. However, the subject is continually growing in breadth and depth, and the reader should augment his study of this (or any other) textbook by a determined assault on the symposium volumes of the *IEEE Symposia on Switching and Automata Theory*, and of the *ACM Symposia on Theory of Computing*, and on journals such as the *Journal of the Association for Computing Machinery*, *Information and Control*, and the *IEEE Transactions on Computers*, which are cited again and again in the bibliography.

CONTENTS

BACKGROUND

1 AN OVERVIEW OF AUTOMATA THEORY

This is the sort of introduction that should be read quickly before the rest of the book, and slowly afterward. I do not attempt to give a detailed tour of the subject matter of this book, nor a survey of those important theorems which this book omits. Rather I try to provide a conceptual framework in which the diverse topics of automata theory may be fitted. So many different descriptions, so many different devices, engage our attention that at times we wonder whether we are getting entangled unnecessarily in idiosyncratic formalizations. While expecting that future research will polish many of the results treated in this book, and reveal some of them as aspects of deeper truths, nonetheless I hope to show that much of the diversity we encounter is desirable, stemming from our need to ask different types of questions about a basically coherent subject matter.

1.1 THE AIMS OF ABSTRACT AUTOMATA THEORY

The *Oxford English Dictionary* defines an *automaton* (plural, *automata*) as "Something which has the power of spontaneous movement or self-motion; a piece of mechanism having its motive power so concealed that it appears to move spontaneously; now usually applied to figures which simulate the actions of living beings, as clockwork mice, etc." Today the computer has replaced the clockwork mouse as the archetype of the automaton; and with it, our emphasis shifts from simulation of motion to simulation of information processing. Automata theory, in its widest sense, might now embrace such diverse activities as the building of a space station's control system or the programming of a computer to play chess. In the theory of *abstract* automata, we are less concerned with the design of automata to do specific tasks, and more concerned with understanding the capabilities and limitations of whole *classes* of automata. One might say, then, that "Automata theory (the qualification abstract is henceforth implied) is the pure mathematics of computer science." Of the several possible interpretations, the one I intend is that, just as much of today's pure mathematics can be seen to have evolved from formalisms suggested by the problems of physics, so automata theory is a branch of mathematics which draws inspiration and intuition from asking questions about biological and electronic computers. The mathematics is pure in that many of the questions are pursued for their intrinsic interest, rather than in the hope of applications. And as answers to these questions accumulate, one is led to look for mathematical generalizations which lay bare the essential logic of the situation, stripped of the details unessential to gaining a general understanding of the processes involved. That we persevere even though some of these details were essential to the "real-world" problem of getting an answer out of a computer by next Tuesday is what makes us—at least on this occasion—automata theorists, rather than programmers or designers of actual circuitry. I like to dream that automata theorists will one day do for computer science what group theorists did for physics. Be that as it may, it must be emphasized that automata theory is *not* to be thought of as the study of a limited number of presently formalized objects. Rather, it is a growing subject which gains richness and power from the intuition one obtains by thinking about information-processing, and the consequent interplay between rigorous mathematics and the search for appropriate formalizations for this intuition.

If automata theory does not yet provide a magical key to the solution of problems of everyday information-processing, it does have the following virtues: (a) it is a fascinating branch of mathematics; (b) it provides a form of mental discipline which will benefit system theorists, computer scientists, and logical designers by giving them a powerful set of languages for setting out their problems, even though it may not provide methods that can be "plugged in"; and (c) certain techniques of automata theory are already directly useful (though, in this book, I shall generally leave this implicit) and, what is more important, automata theory is slowly building up for us a feel for information-processing that will eventually help us do things for which no amount of program-writing could provide the basis. We may note that automata theory has already provided the following:

(1) A characterization of *all* computable functions (e.g., as those computable by Turing machines)—it being now a highly active area of research to find which of the concomitant computations are *practicable*—together with the demonstration that no computer can compute, of an arbitrary computer, whether or not that second computer will ever halt.
(2) The demonstration of *universality*—that there is a computer which can do the job of any other computer provided that it is suitably programmed.
(3) Parsing systems for formal languages, and concomitant automata, which form the basis for a rigorous treatment of compilers for computer languages.

We may expect further progress in automata theory to provide an ever richer *framework* for the solution of practical problems—as the theorist proves theorems of the kind "Every X in the relation R to Y must have property Z," so will the circuit designer learn to check the Z factor each time he builds an R device.

1.2 THE MANIPULATION OF STRINGS OF SYMBOLS

Consider the item displayed on the next line:

$$1\ 0\ 0\ 1\ 1$$

Is it "ten thousand and eleven" or "nineteen in binary notation"? Clearly, it is in fact a string (we use "string" as a synonym for "sequence") of five symbols, of which the first, fourth, and fifth are 1's, while the second and third are 0's. Whether we choose to interpret this string of symbols as a decimal number or binary number, or as something else, depends on our "mental set," on the context. To a binary computer 1 0 0 1 1 is "nineteen"; to a decimal computer it is "ten thousand and eleven." Similarly, the function which places 0 at the end of a string of 0's and 1's is, to a binary computer, "multiplication by two," whereas to a decimal computer, it is "multiplication by ten."

The point I am trying to make, then, is the familiar one that computers are symbol-manipulation devices. What needs further emphasis is that they are thus numerical processors, but *the numerical processing that they undertake is only specified when we state how numbers are to be* ***encoded*** *as strings of symbols which may be fed into the computer, and how the strings of symbols printed out by the computer are to be* ***decoded*** *to yield the numerical result of the computation.*

Our emphasis in what follows, then, is on the ways in which information-processing structures (henceforth called automata) transform strings of symbols into other strings of symbols. Sometimes it will be convenient to emphasize the interpretation of these strings as encodings of numbers, but in many cases we shall deem it better not to do so.

Let us, then, introduce some basic terminology. We shall usually use X to denote the *input alphabet*, the set of symbols from which we may build up strings suitable for feeding into our automaton. The symbol Y will usually denote the *output alphabet*, our automaton emitting strings of symbols from Y.

Given any set A, we shall denote by A^* the set of all finite sequences of elements from A, and shall call the number of symbols, $l(\alpha)$, in a sequence α the *length* of α. For mathematical convenience, we shall include in A^* the *empty sequence* Λ of length 0. We need Λ for the same reason that we had to invent the number 0. Just as it became distinctly unhelpful to write "x with nothing added to it" instead of "$x + 0$," so we prefer to say "input Λ" rather than "no input was supplied." It allows us to state many theorems in general form, without having to treat "no input" as a special case. Given two sequences $\alpha = a_1 \ldots a_n$ and $\beta = b_1 \ldots b_m$ we may *concatenate* them to obtain $\alpha \cdot \beta = a_1 \ldots a_n b_1 \ldots b_m$, and for all α we set $\alpha \cdot \Lambda = \Lambda \cdot \alpha = \alpha$.

Thus X^* will usually denote the set of all input strings to our automaton, and Y^* will indicate a set which includes all possible output strings of our automaton.

Our general notion of an automaton, then, is a device to which we may present a string of symbols from X^*. If and when the machine finishes computing on this string, the result will be an element of Y^*. We say "if" because certain input strings may drive the computer into a "runaway" condition—e.g., endless cycling through a loop—from which a halt is impossible without external intervention (which amounts to changing the input string). This case might correspond to associating with the machine a device which produces an infinite string of elements of Y for each string in X^*—but, in fact, this viewpoint will only be taken in Section 7.2, and so we shall not treat it further in this chapter.

Thus, in a very general form, we may say that automata theory is the study of *partial* functions F: $X^* \longrightarrow Y^*$—that is, ways whereby *some* of the strings in X^* have assigned to them output strings from Y^*, it being understood that for other input strings x, the function $F(x)$ may not be defined at all. However, such a function becomes truly a part of "classical" automata theory only if we can relate it to a *finitely specifiable substrate*—or if we are eager to prove that no such substrate exists for it. Of course, once one has developed a body of theorems, one sees how they can be generalized if the finiteness condition is removed, so this criterion does not cover all of present-day automata theory.

1.3 ON-LINE AND OFF-LINE MACHINES

We have said that automata theory deals with the realization of *partial* functions F: $X^* \longrightarrow Y^*$ by some finitely specifiable substrate. Before we specify in more detail the forms (of which the Turing machine is one) of substrate which have figured most prominently in automata theory, it is useful to distinguish *on-line* machines from *off-line* machines. An on-line machine is one that may be thought of as processing data in an interactive situation—in processing a string it must yield a continual flow of outputs, processing each symbol completely (albeit in a way dependent on prior inputs) before it reads in the next symbol. This means that the corresponding function F: $X^* \longrightarrow Y^*$ must have the following special property:

1 For each nonempty string u of X^* there exists a function F_u: $X^* \longrightarrow Y^*$ such that for every nonempty v in X^*

$$F(uv) = F(u) \cdot F_u(v)$$

that is, the input string u causes the machine to put out the string $F(u)$ and to "change state" in such a way that it henceforth processes inputs according to a function F_u determined solely by F and u. We call a function *sequential* if it satisfies property **1**. If we define $f(\Lambda) = F(\Lambda)$, whereas, for $x \neq \Lambda$, the function $f(x)$ is the substring of $F(x)$ produced in response to the last symbol of x, we see that f: $X^* \longrightarrow Y^*$ allows us to reconstruct F by the formula

$$F(x_1x_2 \ldots x_n) = f(\Lambda)f(x_1)f(x_1x_2) \ldots f(x_1x_2 \ldots x_n)$$

if each $x_1, \ldots, x_n$ is in X. Conversely,

$$f(x_1x_2 \ldots x_n) = F_{x_1x_2 \ldots x_{n-1}}(x_n)$$

Let us define, for any u in X^* the function L: $X^* \longrightarrow X^*$ which simply places u to the left of any string: $L_u(x) = ux$. Then we see that f_u, the function corresponding to F_u, has the simple form

2 $$f_u(x) = fL_u(x)$$

and for a string uv we find, by changing f to f_u, and u to v in **2**, that

$$f_{uv}(x) = f_uL_v(x) = fL_uL_v(x)$$

It thus makes sense to speak of each function f_u for u in X^* (so that $f = f_\Lambda$) as a *state* of the sequential function F—with the input v serving to change state f_u to state f_{uv}, while the output corresponding to state f_u is $f_u(\Lambda) = fL_u(\Lambda) = f(u)$. Thus the evaluation of F may be captured by the input-state-output sequence shown in Table 1.1.

Table 1.1

Input string: Sequence of symbols from X		x_1	x_2	$\ldots$	x_n
State string: Sequence of functions f_u	f	f_{x_1}	$f_{x_1x_2}$	$\ldots$	$f_{x_1x_2\cdots x_n}$
Output string: Sequence of strings on Y^*	$f(\Lambda)$	$f(x_1)$	$f(x_1x_2)$	$\ldots$	$f(x_1x_2\ldots x_n)$

$$\underbrace{f(\Lambda)\ f(x_1)\ f(x_1x_2)\ \ldots\ f(x_1x_2\ldots x_n)}_{F(x_1x_2\ldots x_n)}$$

Three portraits of a sequential machine are shown in Fig. 1.1. Since we are in the habit of considering the first letter of a string to be the leftmost letter (translators of this text into Hebrew, please take care!), it seems appropriate (although perhaps a majority of authors use the opposite convention) to have the input line drawn to the right of the box—so that the first input symbol is the first to reach the box—and the output line drawn to the left of the box—so that the last output symbol is the last to leave the box.

A sequential machine M is specified by three sets, the set X of inputs, the set Y of outputs and the set Q of states together with a next-state function $\delta: Q \times X \rightarrow Q$ and an output function $\beta: Q \rightarrow Y$. If we have $\beta: Q \rightarrow Y^*$, we usually call M a *generalized* sequential machine.

We say that a sequential function F is *finite-state* if there are only finitely many distinct functions of the form fL_u.

Suppose that F has n states, and consider the effect of an input string of n identical inputs, each, say, 0. Then we have a sequence of $n + 1$ states

$$f_\Lambda, f_0, f_{00}, \ldots, f_{0^n}$$

(where 0^n stands for a string of n 0's—i.e., the nth power of 0 with respect to *concatenation*. We may write 0^0 for Λ). Since F has only n states, at least two of the above, say f_{0^i} and f_{0^j} with $i < j$, must be equal, and thus f_{0^n} equals $f_{0^j}L_{0^{n-j}} = f_{0^i}L_{0^{n-j}} = f_{0^{n-(j-i)}}$. Thus

3 If F has only n states, we cannot enter a state of F for the first time after applying n identical inputs.

4 It is clear that if F is finite-state with n states, then every F_u, for u in X^*, is finite-state with $\leq n$ states.

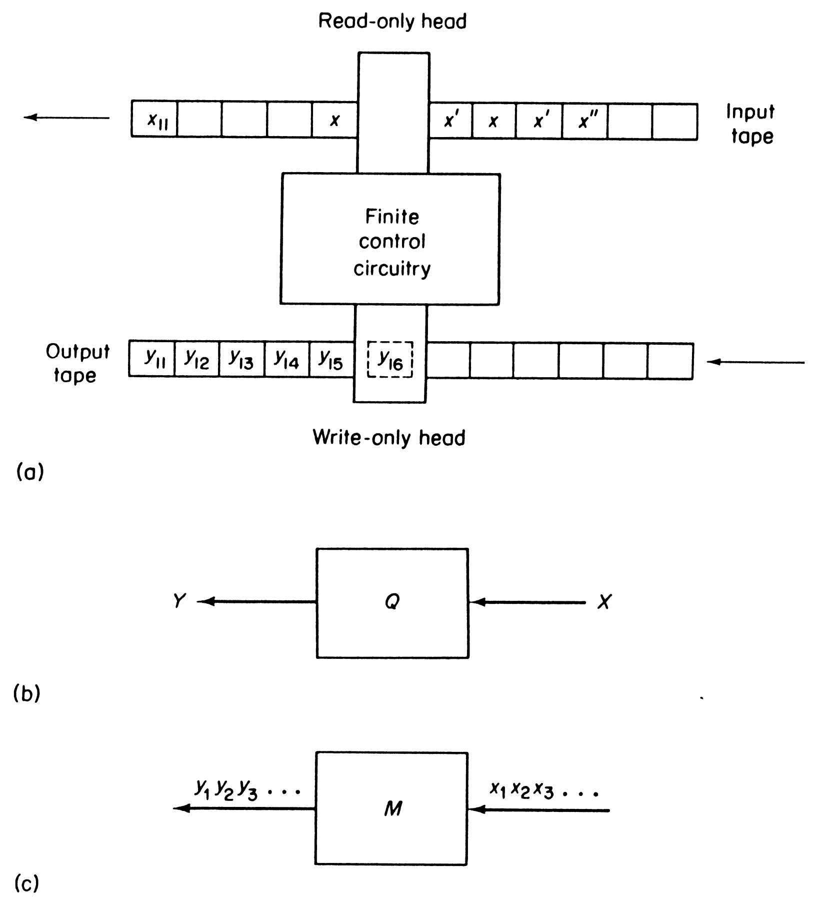

Fig. 1.1

5 EXAMPLE

(i) Let $F: \{0, 1\}^* \rightarrow \{0, 1\}^*$ be the function, with state $f: \{0, 1\}^* \rightarrow \{0, 1\}$,

$$f(x_1x_2\ldots x_n) = \begin{cases} 1 & \text{if an even number of } x_j\text{'s are 1} \\ 0 & \text{if not} \end{cases}$$

Then F is sequential, and is finite-state with only two states $f_0 = f$ and f_1, with the relations

$$f_1 = f_1L_0 = f_0L_1$$
$$f_0 = f_1L_1 = f_0L_0$$

that is, we only change state (parity) if the input is one.

(ii) Let G: $\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0, & 1, & 0, & 1 \end{pmatrix}^* \to \{0, 1\}^*$ be the function

$G\begin{pmatrix} u_1 & u_2 & u_3 & & u_n \\ v_1, & v_2, & v_3, & \dots, & v_n \end{pmatrix} =$ the n lowest-order digits, with lowest-order digit first, of the binary expansion of the product of the binary numbers $u_1u_2 \dots u_n$ and $v_1v_2 \dots v_n$.

Then G is clearly sequential. However, G is *not* finite-state—we shall derive a contradiction from the assumption that G has a finite number n of internal states.

Suppose G had to multiply 2^n by itself fed to the machine as the string $\binom{0}{0}^n \binom{1}{1} \binom{0}{0}^n$ to yield 2^{2n}, that is, the string $0^{2n}1$. This would mean that after it had received its last nonzero input, G would have to print n more symbols, all zeros save for the last. But this would require that, setting $\hat{g} = fL_{\binom{0}{0}^n\binom{1}{1}}$, the state $\hat{g}_{\binom{0}{0}^n}$ is different from each state $\hat{g}_{\binom{0}{0}^j}$, for $0 \le j < n$, contradicting 3 and 4.

(iii) The function H: $\{0, 1\}^* \to \{0, 1\}^*$ with

$$H(x_1x_2 \dots x_n) = x_n \dots x_2x_1$$

is *not* sequential—for instance, $H(01)$ does not begin with the string $H(0)$.

Thus by restricting a machine to process a string one symbol at a time, or to preserve information about prior symbols by the present state from a *finite* set of possible states, we severely limit which functions from X^* to Y^* may be realized.

We are thus led to consider *off-line* machines, which may be simply defined as those functions which need not be sequential. We imagine that the whole string may be presented to the machine before any computation need take place. It is perhaps useful to think of the input string as read into a data structure which the machine may operate upon over a period of time, it usually being assumed that time is quantized, and that only a *finite* portion of the data structure is affected during each unit of time. (This condition will only be relaxed when we consider tesselations in Chapter 10.)

There is a sense, then, in which we may view an on-line computation as treating an input string as distributed in time, whereas an off-line computation treats the string as distributed in space.

A major question, then, is this: Can we formally define a class of machines which can compute all partial functions F: $X^* \to Y^*$ which may be obtained by a well-defined machine when we place a finiteness condition

not upon its memory but only upon its access to that memory? Since the notion of machine is informal in the last sentence, this amounts to finding a precise mathematical definition to replace our intuitive notion of an *effective procedure* for going (not always successfully, since the function may be partial) from a string of X^* to a string of Y^*.

The first candidate for the notion of *effectively computable function* will be that of a *function computable by a Turing machine.* As we develop other theories of computation in this book, we shall see again and again that each computable function we specify can also be computed by a Turing machine. This will bolster our conviction that the notion of *Turing-computable* (and its equivalents) is indeed an adequate formalization of our intuitive notion of effectively computable. However, Turing machines often carry out their computations most inefficiently, and an important task of the automata theorist is to find more efficient automata to compute various classes of functions.

Let us emphasize that we are now considering what effective computations are possible, without placing any bounds on the time or the storage space required to complete the computation. In Chapter 7 we shall turn to the more intricate questions of difficulty or complexity of computation: "Among all the *possible* effective computations, which ones are practicable when we impose certain restrictions on computation time or computer growth?"

At this stage, we had better crystallize the idea of an *effective procedure.* There are certain computations for which there exist mechanical rules, or *algorithms*, e.g., the euclidean algorithm for finding the greatest common divisor of two integers. Certainly, any computation which can be carried out by a digital computer is governed by purely mechanical rules. We say, then, that there exists an effective procedure for carrying out these computations. There are many cases in which we do not really know how to write a program which would cause a given computer to carry out the desired computation, but we do have a strong intuitive feeling that a suitable effective procedure exists.

Abstract automata theory may be said to start with the simultaneous publications of Turing [1936] and Post [1936], who gave independent—and equivalent—formulations of machines which could carry out any effective procedure provided that they were adequately programmed. (Of course, such a statement is informal—we cannot prove that the formally defined class of procedures implementable by Post or Turing machines will be the same as our intuitive hazy notions of effective procedure. Suffice to say that this class has been proved equivalent to many other formal classes, and that no one has produced a procedure which is intuitively effective but cannot be translated into a program for one of their machines.)

The basic idea of the Post and Turing formalisms is as follows (see Fig.

1.2). The machine consists of (i) a control box in which may be placed a *finite* program [i.e., which may be in one of a finite number of states]; (ii) a potentially infinite tape, divided lengthwise into squares (i.e., depending on our choice of mathematical fiction, we may consider the tape as comprising

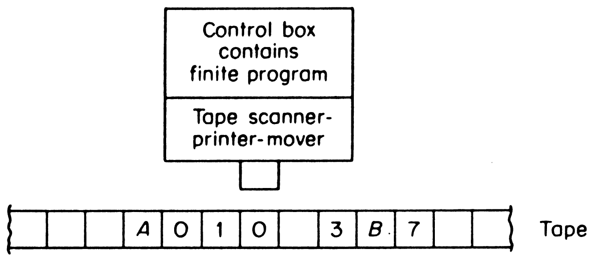
Control box contains finite program
Tape scanner-printer-mover
A
0
1
0
3
B
7
Tape

Fig. 1.2. A Turing Machine

an infinite string of squares of which all but finitely many are blank, or as a finite tape to the ends of which arbitrarily many new squares may be added as required); and (iii) a device for scanning, or printing on, one square of the tape at a time, and for moving along the tape, all under the command of the control box.

We start the machine with a finite string from X^* on the tape, and with a program in the control box. The symbol scanned and the instruction presently being executed (that is, current state of the control box) uniquely determine what new symbol shall be printed on the square, how the head shall be moved, and what instruction is to be executed next (that is, what shall be the next state of the control box). Thus the control box of a Turing machine may be thought of as a finite-state sequential machine whose output can be either a halt instruction, or a print-and-move instruction. *If* and when the machine stops, the result of our computation, a new string from X^*, may be read off the tape.

We shall see that computation more efficient than that of ordinary Turing machines can be obtained simply by allowing the Turing machine to have several tapes—and these not necessarily one-dimensional—each acted on by one or more heads which report back to a single control box which coordinates their printing and moving on their respective tapes. We shall see in Section 4.3 that any job that can be done by such a "polycephalic" Turing machine can be simulated by an ordinary Turing machine—thus reassuring ourselves of the breadth of the notion of a Turing-machine-computable (henceforth: TM-computable) function (which, incidentally, we shall prove in Chapter 6 to be coextensive with the notion of partial recursive function introduced by the logicians in the 1930s as an alternative formalization of the effectively computable functions). Further, we shall show in

Chapter 7 that these polycephalic machines do indeed give us the expected gain in efficiency. We shall also see that they not only have the great virtue of being more efficient in that they take less time to complete a computation, but also are easier to write programs for. With these "polycephalic" machines we shall, in fact, have a realistic model of computers—a virtue we do not claim for the ordinary Turing machine, useful though it is in allowing us to construct a theory of the computable. With these machines we are also made aware that there is no reason to restrict automata theory to functions of the form $F: X^* \to Y^*$. With multidimensional tapes, our automata may as well process planar or higher-dimensional configurations as the linear strings of X^*. However, we shall not pursue this line of study in this book, save for a brief look at pattern recognition in Section 3.2, and in our study of tessellations in Chapter 10.

McCulloch and Pitts [1943] introduced nets of formalized neurons, and showed that such nets could carry out the control operations of a Turing machine—providing, if you will, a formal "brain" for the formal machine which could carry out any effective procedure (cf. Chapter 1 of Arbib [1964]). These nets comprised synchronized elements, each capable of some boolean function, such as "and," "or," and "not." It was his knowledge of these networks that inspired von Neumann in establishing his logical design for digital computers with stored programs, which is of basic importance to the present day. (In 1948, von Neumann [1951] added to the computational and logical questions of automata theory, the new questions of construction and self-reproduction which we shall take up in Chapter 10.)

In 1956 the collection "Automata Studies" (Shannon and McCarthy [1956]) was published, and automata theory emerged as a relatively autonomous discipline. Besides the "infinite-state" Turing-Post machines, much interest centered on finite-state sequential machines, which first arose not in the abstract form of our above discussion, but in connection with the input-output behavior of a McCulloch-Pitts net or an "isolated" Turing machine control box.

Turing's paper [1936] contains a charming "pseudopsychological" account of why we might expect any algorithm to be implementable by a suitable A-machine (his name for Turing machines). We reproduce excerpts from this below. Bear in mind that when Turing wrote this, "computer" meant a human who carried out computations!

> All arguments which can be given are bound to be, fundamentally, appeals to intuition [since the notion of effective procedure is intuitive] and for this reason, rather unsatisfactory mathematically . . . . Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic, the 2-dimensional character of the paper is sometimes used. But such a use is

always avoidable, and I think that it will be agreed that the 2-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e., on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. . . . It is always possible to use sequences of symbols in the place of single symbols. . . . The difference from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. . . . We cannot tell at one glance whether 9999999999 and 99999999999 are the same.

The behavior of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols on squares which the computer can observe at any moment. If he wishes to use more, he must use successive observations. We will also suppose that the number of states of mind which need to be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols . . . . Let us imagine that the operations performed by the computer are split up into "simple operations," which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of those are observed by the computer (possibly with a special order), and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered, [and] . . . without loss of generality assume that the squares whose symbols are changed are always "observed" squares.

Besides these changes of symbols, the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognizable by the computer . . . . Let us say that each of the new observed squares is within L squares of an immediately previously observed square.

The simple operations must therefore include:
(a) Changes of the symbol on one of the observed squares.
(b) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

It may be that some of these changes necessarily involve a change of state of mind . . . . The operation actually performed is determined . . . by the state of mind of the computer and the observed symbols. In particular they determine the state of mind of the computer after the operation is carried out.

We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an [internal state] of the machine. The machine scans B squares corresponding to the B squares observed by the computer. In any move the machine can change a symbol on a scanned square, or can change any one of the scanned squares to another square distant not more than L squares from one of the other scanned squares.

The move which is done, and the succeeding [internal state] are determined by the scanned symbol and the internal state. The machines just described do not differ very essentially from [Turing machines] . . . [and so, a Turing machine] can be constructed to compute . . . the sequence computed by the computer.

We associate with a Turing machine Z a function F_Z: $X^* \rightarrow X^*$ by defining $F_Z(u)$ to be the expression printed on the tape when Z stops, if started in a specified initial state, say q_0, scanning the leftmost letter of the string u—if Z never stops after being so started, $F_Z(u)$ is to be left undefined.

Note that F_Z may be always defined, sometimes defined, or never defined. Trivial examples of the three cases are, respectively, a machine which halts under all circumstances; a machine which halts if it scans a 1, but moves right if it scans any other symbol; and the machine which, no matter what state it is in and no matter what it sees, always moves right.‡

If we think of Turing machines as actual physical devices, it is clear that the input symbols could be configurations of holes on punched tape, patterns of magnetization or handwritten characters, or the like, whereas the states could be that of a clockwork, of a piece of electronic apparatus, or of an ingenious hydraulic device. Such details are irrelevant to our study, and so if there are m inputs in X, we shall feel free to refer to them as $x_0, x_1, \ldots, x_{m-1}$ without feeling impelled to provide further specification; and if there are n states, we shall similarly find it useful to lable them $q_0, q_1, \ldots, q_{n-1}$. I mention this because I want it to be clear that automata theory deals with abstract descriptions of machines, not their implementations. For each such abstract description there are many implementations—depending, e.g., on whether we interpret x_3 as a 0 or a 1, as a pattern of magnetization or a configuration of holes in a punched tape—but it should be clear that, from the information-processing point of view, there is a very real sense in which all these machines may be considered the *same* machine.

Since each Turing machine is described by a finite list of instructions, it is easy to show that we may *effectively* enumerate the Turing machines

$$Z_1, Z_2, Z_3, \ldots$$

so that, given n we may effectively find Z_n, and given the list of instructions for Z, we may effectively find the n for which $Z = Z_n$.

This implies that we can effectively enumerate all TM-computable (that is, Turing machine computable) functions as

$$f_1, f_2, f_3, \ldots$$

simply by setting $f_n = F_{Z_n}$. Such effective enumeration lies at the heart of

‡ This is the John Birch machine. Teachers at more conservative campuses may replace this example with a suitably insidious device which always moves left.

much of our study of the computable in Chapter 4. For example, if we say that f_n is *total* if $f_n(u)$ is defined for all u, we might ask: Is there an effective procedure for telling whether or not f_n is total; e.g., does there exist a total TM-computable function h such that f_n is total if and only if $n = h(m)$ for some m (identifying a string with a suitable number that encodes it)? The answer is "NO," for if such an h existed, we could define f by

$$f(n) = f_{h(n)}(n) + 1$$

Then f would be total recursive, and so $f = f_{h(m)}$ for some m.

Then $f_{h(m)}(m) = f_{h(m)}(m) + 1$, a contradiction!

This is just one example of the many things we can prove to be undecidable by any effective procedure. To say that we cannot effectively tell that f_n is *total* is just the same as saying that we cannot tell effectively whether Z_n will stop computing no matter what tape it is started on. We may thus say that "the halting problem for Turing machines is unsolvable."

A most interesting result of Turing's paper is that there is a *universal* Turing machine, i.e., one which, when given a coded description of Z_n on its tape, as well as the data x, will then proceed to compute $f_n(x)$, if it is defined. This is obvious if we accept that every effective computation may be executed by a Turing machine: For given n and x we find Z_n effectively, and then use it to compute $f_n(x)$, and so there should exist a Turing machine to implement the effective procedure of going from the *pair* (n, x) to the value $f_n(x)$. A proper proof (Section 4.5) takes somewhat longer! The universal Turing machine is the intellectual forebear of today's stored-program digital computer.

1.4 FORMAL LANGUAGES

We have spoken of automata as finitely specifiable substrates for partial functions $F: X^* \to Y^*$. Let us now see ways in which functions may define sets, thus learning to associate classes of sets with classes of automata. We follow the terminology of SCOTT [1967].

We shall say that a subset S of X^* is *decidable* by $F: X^* \to Y^*$ if there exist two distinct elements a and b of Y^* such that $F(u) = a$ if u is in S, whereas $F(u) = b$ if u is not in S. In other words, if we can compute the function F, we have a straightforward method for *deciding* whether or not u belongs to S.

We shall say that a subset S of X^* is *acceptable* by $F: X^* \to Y^*$ if there exists an element a of Y^* such that $F(u) = a$ if u is in S. Note that if F is a *total* function (i.e., defined for all u in X^*), then we may use F to decide

whether or not any u belongs to S. Suppose, however, F is only partially defined, and we set some automaton going to decide u. Then if after a period of time the automaton has not halted, we may not be sure whether it will never halt, so that u is not in S, or whether it will eventually halt, at which time we would know that u belonged to S iff‡ the output of the automaton were a. We have already mentioned that there is no effective procedure to tell of a Turing machine whether or not it will halt.

We shall say that a subset S of Y^* is *generable* by F: $X^* \to Y^*$ just in case S is the *range* of F, that is, $S = F(X^*) = \{F(u) | u \text{ is in } X^*\}$.

We shall usually speak of subsets of X^* or Y^* as *languages*. Thinking of X, say, as the vocabulary, we think of the strings of X^* as the possible utterances in the vocabulary, with the strings of the given subset as being, in some sense, grammatical or well-formed. We shall associate different *classes* of languages with different *classes* of automata and explore the formal properties of these associations, leaving it for texts on linguistics to assess what relevance such studies have to the structure of the real languages of human discourse.

We may note that if we consider functions $f: X^* \to Y$ corresponding to sequential functions $F: X^* \to Y^*$ which are *finite-state*, then we get the same class of languages whether we use the functions as deciders or acceptors. We speak of languages in this class as *finite-state languages*. The reader may wish to prove, after he has read Section 3.3, that they are the same class of languages as those *generated* by finite-state (generalized) sequential functions $F: Z^* \to X^*$.

Now, we have already suggested that the Turing-computable functions $F: X^* \to X^*$ correspond to the intuitive notion of effectively computable functions. Thus the classes of sets which are decided, accepted, or generated by them should include the classes of sets decided, accepted, or generated by other finitely specified automata. (I make the proviso "finitely specified," since the mathematical automata theorist is under no compulsion to restrict his attentions to this case. Once he understands this primary focus of automata theory, he may then find it worthwhile to seek nonfinite generalizations.)

We shall see in Chapter 4 that the class of functions *decided* by Turing-computable functions—the *recursive sets*—is a *proper* subclass of the class *accepted* by Turing-computable functions—the *recursively enumerable sets.* However, the sets *generable* by Turing machines are precisely the recursively enumerable sets.

One of the centers of study in the theory of formal languages is that of *closure properties*. We may ask whether or not a class is closed under comple-

‡ 'iff' is the standard abbreviation for 'if and only if' and will be used throughout.

mentation—i.e., whether, for every subset R of X^* which belongs to the class, it is also true that the complement $X^* - R$ belongs to the class. It is clear that every class of *decidable* sets is closed under complementation—we just interchange the role of a and b in the definition of the set. Similarly, one may prove that any reasonable class of generable sets is closed under the formation of finite unions—we just use the first letter of the input to switch us to the computation for generating an element of the appropriate term of the union.

We shall see in Section 3.3 not only that the finite-state languages are closed under union, set product (which replaces the sets E and F by the set $E \cdot F$ of all strings obtainable as a string of E followed by a string of F), and iteration (the operation that replaces a set E by the set E^* of all strings obtainable by concatenating a nonnegative number of strings from E)—as well as complementation and intersection—but also that these first three operations serve to characterize the finite-state languages. In fact, a set is a finite-state language iff it can be built up from finite sets, using a finite number of applications of $\cup$, $\cdot$, and $*$ operations. This will be our first taste of machine-independent characterizations of a class of languages which may also be characterised by a class of machines.

The beginnings of a hierarchy of classes of machines—with lower end defined by finite-state sequential machines, and upper end defined by the Turing machines—defines the beginning of a hierarchy of classes of languages—with lower end defined by finite-state languages, and upper end defined by the recursively enumerable sets, with the recursive sets falling properly in between. A major aim is to fill out in more and more detail the structure of these two hierarchies.

We shall see in Chapter 5 that languages can be defined in a machine-independent way by formalized "grammars," and shall then be led to study two natural classes of languages, the so-called context-sensitive and context-free languages. We shall find that the finite-state languages are a proper subclass of the context-free languages, which are similarly related to the context-sensitive languages, which in turn form a proper subclass of the recursive sets. But this is not all. We shall see that we may associate with these languages two classes of automata—each obtained by imposing natural restrictions on Turing machines—the push-down automata with the context-free languages, and the linear-bounded automata with the context-sensitive languages. And so it goes on, the literature now abounding with new classes of automata, intermediate between the finite automata (i.e., finite-state sequential machines) and the Turing machines. It is too early to provide an overview of all these variants, and to judge which are worthy of continued study and which mutations are nonviable—but the material presented in Part II of this book should more than suffice as background for the reader who wishes to approach this literature and judge for himself.

1.5 HIERARCHIES AND DIVERSITIES

A hierarchical classification of machines does not arise merely in connection with the study of formal languages.

We may impose subhierarchies on the finite automata (i.e., finite-state sequential machines) by asking how many components of a given kind are required in the construction of a given machine, as we do in Section 2.2, where the components are boolean switching elements with a fixed number of input lines, or in Section 8.6, where the components are finite automata whose inputs all serve to permute the set of states. These are studies which lay the foundations for really practical analyses of cost of realization of abstract descriptions of automata, at the same time as they give us insight into the way the *structure* of the finitely specifiable substrate *must* vary as we alter the nature of the input-output function it is to embody.

We may define subhierarchies of the Turing-computable functions in terms of inductive definitions of classes of functions, often far removed from machine definitions—in the manner of recursive function theory—and yet, as we shall see in Chapter 6, the subclasses so defined often correspond to subclasses of the Turing machines, and therein lies their interest to automata theorists.

Or we may seek to restrict our machines by limiting the time or tape space they may use in completing a computation, and then ask where in the complexity scale so induced lie the solutions of various problems. For instance, in Chapter 7 we shall give a formal proof of the old saw that "two heads are better than one" by proving that a Turing machine with two heads on one tape can recognize whether or not a string is a palindrome (i.e., reads the same backward and forward: "ABLE WAS I ERE I SAW ELBA," and the like) in a time that only increases linearly with the length of the string, while the time taken by a one-head machine will, for nearly all strings, go up with the square of the length of the string.

The reader should begin to understand why automata theory has so many different devices. Automata may be viewed in many different ways, and each way gives rise to a different set of questions. No matter how well we may evolve a unified theory of automata, these differences will remain, and the interrelations of these differing approaches will provide fertile fields for study. Just as much of the excitement of automata theory is generated by the attempt to capture diffuse aspects of reality in a convenient formalization, so there is a richness of insight to be gained in cross-formalization studies, in which we try to understand the mismatches between formalizations, and why advantages of a formalization in one context become disadvantages in another. Let us close, then, by listing some of the different ways in which we may consider automata, whether or not we have already discussed them in this chapter. We may consider automata as:

Constructs: Given a collection of components, we may ask questions of *synthesis*—can a given behavior be realized by an interconnection of these components, and if so, how?—and of *analysis*—what is the behavior of a given interconnection of the components?

Programs: Given a collection of input and output commands, data manipulation, and branch-on-test instructions, we may study the computations which can be carried out by executing a program written in terms of these commands.

Functions: We may view automata as either on-line or off-line devices for transforming input strings in X^* (not always successfully) to output strings in Y^*, thus yielding a (possibly partial) function $F: X^* \to Y^*$.

Languages: An automaton may be used for deciding whether or not input strings belong to a language, or accepting those strings which belong to a language, or generating a language as the set of its output strings. With languages, as with functions, we shall be led by our machine-theoretic investigations to study machine-independent questions, such as closure properties, which will lead us to other characterizations.

Logics: We may study devices for generating proofs of theorems from a given set of axioms, using given rules of inference; or a device for deciding whether or not a string is a theorem in a given formal system. This study may be formally like that of languages, but with the interpretation "This is a theorem" replacing the interpretation "This is grammatical."

Dynamic systems: Here the primary emphasis is on the state of the system, and the ways in which inputs may be chosen to control this state—outputs thus being relegated to a secondary role. This is the approach which links automata theory to mathematical system theory.

Algebraic systems: Here we emphasize the way in which, e.g., finite automata may be considered as a generalization of semigroups, thus enabling us to ask new questions about algebra, yet at the same time applying algebraic techniques to questions about automata. We can also study generalizations of automata suggested by generalizations of the algebraic concepts we apply.

We study the interplay between machine, language, and algebraic characterizations. We see how concepts change as we go from deterministic to possibilistic (nondeterministic yet not probabilistic) to probabilistic modes of operation. We extend our study of computation to deal with problems of construction. We impose hierarchies upon our constructs by imposing

constraints of time, space, or complexity. And we ask how our formalizations compare with the complexities of real information-processors. The result is a growing, open-ended, mathematically sophisticated yet intuitively appealing abstract theory of automata.