AUTOMATA THEORY

AUTOMATA AS MODELS FOR COMPUTATION

The theory of automata is a fundamental theory in computer science. It originated mainly in the 1930s when A. M. Turing (1) developed his mathematical model for the precise definition of functions that can at least in principle be computed by a mechanical device (machine). The idea of a machine that can perform arithmetical computations is much older and was motivated not only by practical purposes but also by philosophical questions concerned with the abilities of the human brain.

Turing analyzed the process of a computation that a human being performs. He regarded it as a purely symbol-manipulating task based on a few simple rules that are applied over and over again. This analysis led to a mathematical machine model, the Turing machine, that is on one hand surprisingly simple and on the other hand very powerful. The thesis of Church and Turing states that exactly those functions that we intuitively believe to be computable are the functions that can be computed on a Turing machine. Thus the model of the Turing machine is one way to define the notion of a computable function or that of an algorithm. A Turing machine can also be regarded as an algorithm that computes a special function.

Having defined the computable functions, it was possible to give examples of functions that are easy to specify but probably not computable, as for example the decision as to whether a given Turing machine will eventually halt for a given argument (halting problem) or the famous tenth problem of David Hilbert (2). The same machine model also was used to define the time and the space it takes for a given algorithm to compute the values for the arguments. These questions opened the wide new area of computational complexity. As a result, algorithms could be classified according to the amount of time and space they consume. It turned out that there are many functions that only have algorithms that need so much time (or space) that they are not feasible, that is, not computable from a practical point of view. The most famous unsolved problem in theoretical computer science is concerned with the question of whether a large class of practically important functions (the NP-complete decision problems) can ever be computed within reasonable (polynomial) time and space bounds on a deterministic machine (the P = NP problem) (3).

A machine, or *automaton,* is an abstract mathematical object that could in principle be built with mechanical, electronic, or other components of known technology. Thus automata constitute the mathematical basis for the construction of electronic digital computers and many other modern information-processing devices. An automaton is a system that has discrete input, output, and state spaces and whose behavior is not described by differential equations but with methods of universal algebra and logic. An automaton manipulates a finite set of symbols using a finite set of simple rules. The theory investigates what automata can do if they are allowed finite (or even countably infinite) sequences of single steps.

The Turing machine is an archetype of the models that are encountered in the theory of automata. Many modifications (restrictions and generalizations) have been investigated. In this article we do not give a complete overview of all the different types of automata that have been the subject of research; for further reading refer to (4,5). Instead we concentrate on a few models that play an important role in different fields of electrical engineering. We introduce the main questions and results of the theory as well as its practical applications in information technology.

For a more formal definition of automata and related concepts we need a few mathematical notions. We assume that the reader is familiar with the concepts and notations of sets, functions, and relations. An introductory textbook on discrete mathematics or computer science may be a useful supplement. We will use the artificial word "iff" as an abbreviation for "if and only if." By $\mathbb{N} = \{0, 1, 2, . . .\}$ we denote the set of *natural numbers* including zero and for $m \in \mathbb{N}$ we define $\boldsymbol{m} = \{0, 1, . . ., m - 1\}$ to be the set of the first m natural numbers. An *alphabet* is a finite set $\Sigma = \{\sigma_1, . . ., \sigma_n\}$ of *symbols.* A finite sequence $x_1x_2 . . . x_k$ of symbols ($x_i \in \Sigma$, $k \in \mathbb{N}$) is called a *word* of *length* k. We include the case $k = 0$ and say that there is a (unique) word of length 0, which will be called the *empty word* and will be denoted by ϵ. The set of all finite words that can be formed with symbols from Σ including the empty word ϵ will be denoted by Σ^*.

The set Σ^* allows for a very simple binary operation called *concatenation*. If $u = x_1 \ldots x_k$ and $v = y_1 \ldots y_m$ are words in Σ^* of length k and m, respectively, then we define $uv = x_1 \ldots x_k\, y_1 \ldots y_m$ as the word of length $k + m$ that is simply the juxtaposition of the two words. The empty word has no effect (is *neutral*) under concatenation: $u\epsilon = u = \epsilon u$. It is easy to see that concatenation is an *associative* operation: $u(vw) = (uv)w$.

A (formal) *language* is a subset $L \subseteq \Sigma^*$ of words over a given alphabet. If L and N are subsets of Σ^*, then we can define the *product* $LN \subseteq \Sigma^*$ by $LN = \{uw \in \Sigma^* \mid u \in L \text{ and } w \in N\}$. LN contains all words that are composed of a first part taken from L and a second part taken from N. So we also can define L^k for $k \in \mathbb{N}$ by $L^0 = \{\epsilon\}$ and $L^{k+1} = L^kL$. The *iteration* (or Kleene star) L^* of a language $L \subseteq \Sigma^*$ is defined as: $L^* = \bigcup_{k\in\mathbb{N}} L^k$ and consists of all finite sequences of words taken from L and concatenated into one new word. If Σ and Γ are alphabets, then a relation $R \subseteq \Sigma^* \times \Gamma^*$ is called a *word relation,* and if it is a partial or total function, it is called a *word function,* and we will denote it as usual by $f\colon \Sigma^* \to \Gamma^*$.

An automaton A is a device that in the most general case computes a word relation $R_A \subseteq \Sigma^* \times \Gamma^*$, thus relating input sequences to output sequences. The relation $R_A \subseteq \Sigma^* \times \Gamma^*$ is called the *behavior* of the automaton A (Fig. 1). A in general is also called a *transducer.* If the set of output sequences that A computes contains at most two elements, then A is called an *acceptor.* In the latter case we may regard R_A as a relation $R_A \subseteq \Sigma^* \times \mathbf{2}$, and then A defines the language $L_A = \{u \in \Sigma^* \mid (u, 1) \in R_A\}$ (Fig. 1). In automata theory acceptors are used to define languages or analyze their structure and transducers more generally are used to define or realize input–output relations.

An automaton has a finite *local state space* Q and a *global state space* K that may be regarded as a model for the total *memory* of the automaton. Global states are also called *configurations.* The *dynamics* of an automaton is a relation $\Delta \subseteq K \times K$ that specifies for each configuration a set of possible *successors.* The dynamics is based on a local rule that we will explain later. If Δ is a partial function, then the automaton is called *deterministic;* otherwise it is *nondeterministic.* Further, we have functions $\mathsf{in}: \Sigma^* \to K$, $\mathsf{out}: K \to \Gamma^*$, and $\mathsf{final}: K \to \mathbf{2}$. The function in maps the input sequences into configurations, out maps configurations to output sequences, and final is a predicate that classifies certain configurations as final. A configuration $c \in K$ is *final* iff $\mathsf{final}(c) = 1$ (Fig. 2).

A finite *computation* of the automaton A is a finite sequence $c_0c_1 \ldots c_m$ of configurations ($c_i \in K$) such that for $0 \le i < m$, c_{i+1} is a successor of c_i—formally, $(c_i, c_{i+1}) \in \Delta$. Mathematically Δ^* is the reflexive and transitive closure of the relation Δ, and $(c, c') \in \Delta^*$ is equivalent to the existence of a finite computation with $c = c_0$ and $c' = c_m$. A finite computation $c_0c_1 \ldots c_m$ is *successful* if c_m is the first final configuration. An infinite sequence $c_0c_1 \ldots c_m \ldots$ of configurations ($c_i \in K$) such that for all i we have $(c_i, c_{i+1}) \in \Delta$ is an *infinite* computation (see Fig. 2).

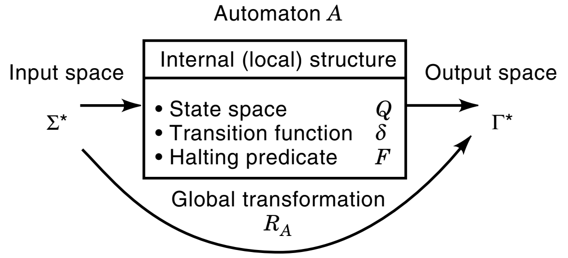

Figure 1. A general system.

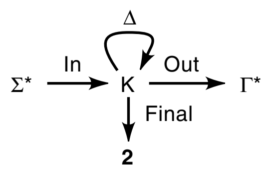

Figure 2. Computation, configuration.

Given an input sequence $u \in \Sigma^*$, we get an initial configuration $c_0 = \mathsf{in}(u)$. If there exists a finite computation $c_0c_1 \ldots c_m$, then we apply the function out to get the output $\mathsf{out}(c_m) \in \Gamma^*$. But it is also possible that, starting at c_0, we never end up in a final configuration. In this case the automaton produces an infinite computation and we say that the automaton does not stop. So for any input sequence $u \in \Sigma^*$ we get a (possibly empty) set of resulting output sequences, and in this way the automaton A defines the input–output relation $R_A \subseteq \Sigma^* \times \Gamma^*$.

Figure 1 shows first a very general structure of an automaton as a special case of general systems (6). The components of the internal structure and the way the global state or configuration is defined depend on the type of automata and will be described in more detail for the different machines in the sequel.

LANGUAGES, GRAMMARS, AND AUTOMATA

Closely related to the theory of automata is the theory of *formal languages.* We have seen that a formal language is just a subset $L \subseteq \Sigma^*$ of the set of all words (finite sequences) built from a finite alphabet Σ. An acceptor is a machine that can define such a language as the set of all sequences that it accepts. The behavior of an acceptor is a formal language, but not all formal languages can be defined by an acceptor with finite local transition rules. We will see later that different types of automata accept different classes of languages and these classes form a hierarchy.

The concept of a formal *grammar* yields another model for the finite characterization of languages. A grammar is a finite set of rules that generate certain words over an alphabet Σ and thus also defines a formal language. We want to introduce the concept of a formal grammar and the way a grammar defines a language, because of the intimate relation of grammars and automata.

A grammar is a special case of the more general *semi-Thue system,* which we describe first. The idea of a semi-Thue system is to specify a finite set of rules that locally manipulate sequences over an alphabet V. A *rule* is an ordered pair $(u, v) \in V^* \times V^*$, and we say that a sequence $x \in V^*$ is *transformed* to $y \in V^*$ in a single step by applying the rule (u, v) iff x has a partition into three subsequences $x = x'ux''$ such that $y = x'vx''$. So applying the rule (u, v) to x means finding a subsequence u (i.e., the left-hand side of the rule) within x and then replacing u by the right-hand side of the rule, namely v. This local manipulation is quite similar to the search-and-replace operation of a word processor or text editor.

If P is a finite set of rules, then we define the *one-step derivation* relation that relates pairs of V^* as follows: $x \Rightarrow y$ iff there is a rule $(u, v) \in P$ such that y is the result of applying the rule (u, v) to x. We extend this relation to its so called *reflexive* and *transitive closure* $\Rightarrow^* \subseteq V^* \times V^*$ by defining $x \Rightarrow^* y$ iff (1) there is a finite sequence of one-step derivations $x \Rightarrow x^{(1)} \Rightarrow x^{(2)} \Rightarrow \cdots \Rightarrow x^{(n)} \Rightarrow y$ that transforms x into y or (2) $x = y$. The sequence $x \Rightarrow x^{(1)} \Rightarrow x^{(2)} \Rightarrow \cdots \Rightarrow x^{(n)} \Rightarrow y$ is called a *derivation* of y from x. A rule $(u, v) \in P$ is also simply denoted as $u \rightarrow v$. Given a word $w \in V^*$, we denote the set of all words $x \in V^*$ that may be derived from the *initial* word w by $L_w = \{x \in V^* \mid w \Rightarrow^* x\} \subseteq V^*$. Thus P and w together define a language over the alphabet V.

A *grammar* is a semi-Thue system where the alphabet V is subdivided into two disjoint alphabets N and T. The elements of N are called *nonterminal* and those of T are called *terminal* symbols. So $V = N \cup T$ is the set of all symbols of the grammar and $N \cap T = \emptyset$. The initial word is a fixed symbol $S \in N$. A *grammar* is a structure $G = (N, T, S, P)$ where N and T are disjoint finite alphabets, $S \in N$ is the *initial symbol,* and $P \subseteq V^* \times V^*$ is a finite set of *rules.* A word $x \in V^*$ that can be derived from S is called a *sentential form* of G, and if the sentential form only consists of terminal symbols ($x \in T^*$), then x belongs to the language defined by G. So G defines the language $L_G = \{x \in T^* \mid S \Rightarrow^* x\}$.

A language that can be generated by a grammar in this way is referred to as being of *Chomsky type* 0. It is important to know that there exist many formal languages that cannot be generated by a grammar. If we are given a grammar G and a word $w \in T^*$, it is in general a difficult task to find a derivation for w and thus to prove that $w \in L_G$. An algorithm that can perform this task is called a *syntax analysis* algorithm. Efficient algorithms for syntax analysis are only available for special classes of grammars. This is the reason that programming languages are defined by grammars of a special form (context-free grammars).

HIERARCHIES OF LANGUAGES AND AUTOMATA

When the form of the rules is restricted, we get special types of grammars. Here we only want to mention two such types. A grammar $G = (N, T, S, P)$ is called *context-free* iff $P \subseteq N \times V^*$. This means that the rules have just one nonterminal symbol on the left-hand side. As a consequence it is very easy to find the left-hand side of a rule within a word and then simply replace it by the right-hand side of the rule. Given nonterminal symbols may be replaced independently and in arbitrary order. This makes it easier to find derivations for a given word. A language $L \subseteq T^*$ is called context-free iff there exists a context-free grammar G that generates L.

Example. Consider the context-free grammar G that consists of $N = \{S\}$, $T = \{[, (,],)\}$ (a set of two different kinds of opening and closing brackets), and the rules $S \rightarrow (S)$; $S \rightarrow [S]$; $S \rightarrow SS$; $S \rightarrow \epsilon$. Here is a derivation for the correct bracket structure [()]():

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow [S](S) \Rightarrow [(S)](S) \Rightarrow [()]()$$

The language L_G consists exactly of the well-formed bracket structures with two different types of brackets. This language is also known as a Dyck language and is denoted by D_2.

For a context-free grammar a derivation may also be represented by a *tree* where the nodes are labeled with the symbols of the grammar or the empty word. The root of the tree is labeled with the initial symbol, and if a node is labeled with a nonterminal symbol $X \in N$ and in one step X is replaced by the right-hand side of a rule $X \rightarrow v_1v_2 \ldots v_k$, then the node has exactly k successor nodes labeled with $v_1, v_2, \ldots, v_k$. If the right-hand side of a rule is $\in$ (empty word), then we use one successor node labeled with $\in$. A node labeled with a terminal symbol has no successor. Such a tree is called a *derivation tree.* The derivation tree for the above example is given in Fig. 3.

A special case of context-free grammars are the *right-linear grammars* where the rules have the special form $X \rightarrow t_1 \ldots t_2Y$ or $X \rightarrow \epsilon$, where X, Y are nonterminal symbols and $t_1t_2 \ldots t_k$ is a sequence of terminal symbols. So in a derivation step we always replace the single nonterminal symbol that is on the right edge of the given word. In this case the derivation tree degenerates to a linear structure (sequence). A language $L \subseteq T^*$ is called right-linear if there exists a right-linear grammar G that generates L. We will see that context-free grammars can generate languages that cannot be generated by any right-linear grammar. So the generative power of context-free grammars is greater than that of right-linear grammars.

Context-free grammars are very important for the syntactic definition of programming languages. They are often represented in the so-called *Backus Naur Form* (BNF) or Extended BNF (EBNF), which are often used for describing the syntax of programming languages. [See Ref. (7).] The class of right-linear languages is also called the class of *regular languages.* Regular languages play an important role not only in programming languages but also in the definition of text patterns for text-processing algorithms.

In the theory of automata and formal languages it is shown that for the special types of grammars there exist special types of automata that accept exactly the languages that can be generated by the grammars of a given type. So the theory establishes on one hand a hierarchy of classes of grammars of different type and on the other hand a hierarchy of classes of types of automata; both define the same hierarchy

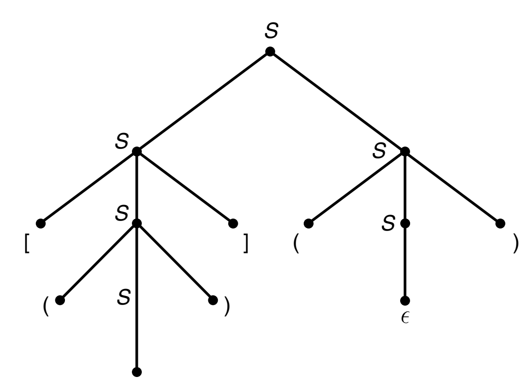

Figure 3. Derivation tree for the word [0]0.

of classes of languages. The most famous such hierarchy is the *Chomsky hierarchy,* which defines four classes of languages in the order of nontrivial class inclusion: type 3 (regular languages), type 2 (context-free languages), type 1, and type 0. We have defined the languages of the types 0, 2, and 3, and we will concentrate on the types of acceptors that accept just those languages. The languages of type 1 are defined by so-called *context-sensitive* grammars or by linear bounded automata (which we only mention here).

TURING MACHINES

The Model

Informally, a *Turing machine* consists of a control unit, a read–write head, and an infinite tape; see Fig. 4. The tape is divided up into cells, and each cell contains exactly one symbol of a given alphabet. An empty cell is represented by the special *blank* symbol #. Only a finite number of cells contains symbols unequal #. A Turing machine can execute the following operation on the tape: reading the cell of the tape to which the read–write head points, replacing the content of this cell by a symbol of the tape alphabet (including the blank), and moving the read–write head one cell to the left or to the right.

A Turing machine is defined as a structure $M = (\Sigma, \Gamma, Q, \delta, \#, q_0, F)$ where $\Sigma \subseteq \Gamma$ is the *input alphabet,* Γ is the *tape alphabet* including the blank symbol #, Q is the finite set of *states,* $q_0 \in Q$ is the *initial state,* $F \subseteq Q$ is the set of *final states,* and $\delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ is the *local transition relation.* If δ is a functional relation (partial function $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$), then the Turing machine M is called *deterministic,* and otherwise *nondeterministic.*

A transition $(q, a, p, b, d) \in \delta$ with $d \in \{L, R\}$ has the following interpretation: If M is in state q, the read–write head reads a symbol a, then M replaces this symbol a by b and moves the read–write head one cell to left (L) or right (R). A *configuration* of M can be described as an element of $\Gamma^*Q\Gamma^*$. In more detail, let $w_1 \ldots w_i q w_{i+1} \ldots w_n$ be the current configuration of M, that is, M is in state q, the read–write head reads w_{i+1}, and the tape contains $w_1 \ldots w_i w_{i+1} \ldots w_n$ and is blank otherwise. The *dynamics* of M is defined as follows:

1. If $(q, w_{i+1}, p, b, R) \in \delta$ and $i < n - 1$, then $w_1 \ldots w_i b p w_{i+2} \ldots w_n \in \Delta(w_1 \ldots w_i q w_{i+1} \ldots w_n)$. In the case of $i = n - 1$ (i.e., the read–write head is at the rightmost position), the tape will be enlarged at the right end by one cell containing the blank symbol, and $w_1 \ldots w_n q\# \in \Delta(w_1 \ldots w_{n-1} q w_n)$.
2. If $(q, w_{i+1}, p, b, L) \in \delta$ and $i > 0$, then $w_1 \ldots w_{i-1} p w_i b w_{i+2} \ldots w_n \in \Delta(w_1 \ldots w_i q w_{i+1} \ldots w_n)$. In the case of $i = 0$ (i.e., the read–write head is at the leftmost position), the tape will be enlarged at the left end by one cell containing the blank symbol, and $q\#w_1 \ldots w_n \in \Delta(q w_1 \ldots w_n)$.

A *start configuration* is given if the Turing machine starts in initial state q_0, the read–write head is at the leftmost cell, which contains a blank symbol, and the cells to the right contain the input word, that is, $\mathsf{in}(u) = q_0\#u$. A *final configuration* has been reached if the Turing machine is in a final state,

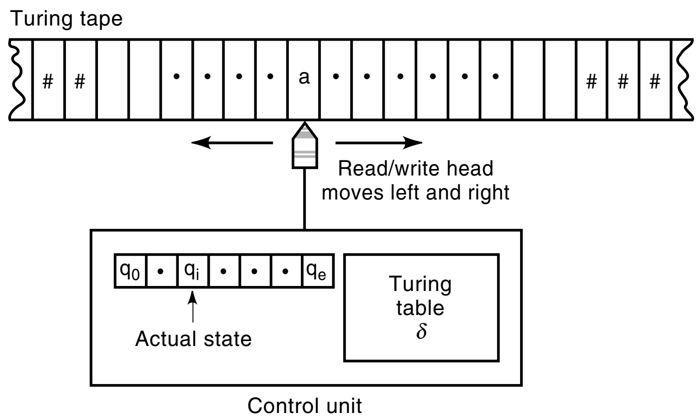

Figure 4. Turing machine.

that is, $\mathsf{out}(vqu) = vu$ and $\mathsf{final}(vqu) = 1$ for $v, u \in \Gamma^*$ iff $q \in F$.

The *accepted language* of a Turing acceptor M is defined as $L(M) = \{w \in \Sigma^* \mid$ there exists a successful computation of M for input $w\}$.

The model of deterministic Turing machines is exactly as powerful as the model of nondeterministic Turing machines; that is, for each nondeterministic Turing machine M one can construct a deterministic Turing machine M' simulating M.

The intuitively *computable functions* are exactly the same as the functions that are computable by Turing machines. We will only consider partial functions from natural numbers to natural numbers. Each natural number can be represented over the alphabet {1} (unary representation), in which $i \geq 0$ is represented by 1^{i+1}. Now a Turing machine computes a function $f(m) = n$ iff the machine starts with a configuration $q_0\#1^{m+1}$. After reaching a final configuration, the tape content represents the computation result, i.e., 1^{n+1} is on the tape. If a function has more than one argument, then the arguments are separated by a special symbol (e.g., 0).

Example. The Turing machine $M = (\Sigma, \Gamma, Q, \delta, \#, q_0, F)$ where $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \#\}$, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $F = \{q_4\}$, and δ consists of the following tuples:

$$\begin{array}{lll} (q_0, \#, q_0, \#, R), & (q_0, 1, q_0, 1, R), & (q_0, 0, q_1, 0, R), \\ (q_1, 1, q_1, 1, R), & (q_1, \#, q_2, \#, L), & \\ (q_2, 1, q_3, \#, L), & (q_2, 0, q_4, \#, L), & \\ (q_3, 1, q_3, 1, L), & (q_3, 0, q_4, 1, L) & \end{array}$$

computes the addition function +. The transition relation can also be written in form of a so-called Turing table:

	0	1	#
q_0	$q_1, 0, R$	$q_0, 1, R$	$q_0, \#, R$
q_1		$q_1, 1, R$	$q_2, \#, L$
q_2	$q_4, \#, L$	$q_3, \#, L$	
q_3	$q_4, 1, L$	$q_3, 1, L$	
q_4			

where each table entry shows the possible action with respect to a state and a tape symbol. The Turing machine moves to the rightmost 1, replaces it by #, then moves back to the left and searches for the separation between the two arguments, replaces the symbol 0 by 1, and halts.

Universal Turing Machine

Turing designed a single fixed machine, a *universal Turing machine,* to carry out the computations of any Turing machine. The universal Turing machine is nothing but a *programmable* Turing machine that, depending on its input program, can simulate other Turing machines. A program of a Turing machine represents a description of the Turing machine to be simulated. Since every Turing-machine definition is finite, it is possible to encode the Turing table (e.g., in binary code). The resulting coded Turing machine is put onto the tape of the universal Turing machine together with the encoding of the concrete input word w of the Turing machine M to be simulated. The initial configuration of the universal Turing machine is q_0# coded M# coded w. Now the universal Turing machine simulates the activation of M on w on the basis of the coded Turing table of M. In a final configuration the right part of the tape, initialized with w, contains the computed result.

In this sense the universal Turing machine is an idealized conception of existing programmable computers. Surprisingly, only seven states and four symbols are sufficient to define a universal Turing machine; see Ref. 8.

Noncomputable Functions

A famous result of theoretical computer science is that the above-mentioned *halting problem* for Turing machines is undecidable, that is, the question "Given a Turing machine M and an input w, does M halt when started on w?" cannot be answered. More precisely, there does not exist a Turing machine that always stops and answers the above question with 0 (no) or 1 (yes) for each input M, w.

A rather unsatisfying argument to that effect is the consideration that after each fixed number of steps of M, we can decide whether M is in a final configuration or not. But we cannot conclude from a nonfinal configuration that M will never halt, because we do not know what will happen in the future. Is it possible to reach a final configuration or not?

To show that the halting problem for Turing machines is undecidable, we use a more complicated construction. Suppose that there exists a Turing machine H that solves the halting problem. Similarly to universal Turing machines, the machine H starts with an input consisting of the representations (coding) of M and an input word w and outputs 1 if M halts on w and outputs 0 otherwise. See Fig. 5.

Now we can construct a Turing machine H' from H by adding transitions before H enters a final configuration. These

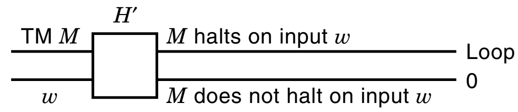

Figure 6. Undecidability of the halting problem (step 2).

additional transitions define an infinite loop as in Fig. 6. Since each Turing machine can be suitably encoded, the Turing machine H' can be applied to its own description twice, as program and as data; see Fig. 7. Now we have the situation that H' halts on input H' iff H' does not halt on input H', a contradiction. Therefore there does not exist any Turing machine that solves the halting problem.

Generalizations of the Turing Machine

Many generalizations of the Turing-machine model have been considered with respect to tapes infinite on only one side and to the numbers of tapes and read–write heads, dimensions of the tape(s), and so on. These extensions do not really increase the power of the original model; see Ref. 7.

Complexity Hierarchies

Different Turing machines can compute the same function f. To compare and to classify different Turing machines computing f we need an appropriate measure. The amount of resources needed to perform a valid computation of a Turing machine is such a measure. In more detail, the number of steps performed during a computation by a Turing machine is called the computation *time,* and the number of cells on the tape required for the computation is called the computation *space.* The *time complexity* $T: \mathbb{N} \to \mathbb{N}$ of a Turing machine M can be defined as follows: for each input word w of length n, the number of transitions of M before halting is limited by $T(n)$. In a similar way, the *space complexity* $S: \mathbb{N} \to \mathbb{N}$ of a Turing machine M can be defined: for each input word w of length n, the number of cells on the tape used by M before halting is limited by $S(n)$. Both complexity measures can be applied to nondeterministic and deterministic Turing machines and lead to various *complexity hierarchies* (3,7).

One famous unsolved problem is the P = NP *problem,* which asks: Is it possible to simulate each nondeterministic Turing machine with polynomial time and space complexity by a deterministic Turing machine with polynomial time and space complexity?

Since the Turing machine model defines the subject algorithm, all results gained about complexity measures of Turing machines can be carried over to algorithms. This implies that there are (theoretically) computable functions that are not practically realizable because of their high complexity. Today, problems of exponential complexity are regarded as practically unsolvable.

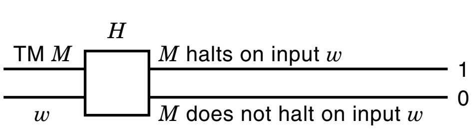

Figure 5. Undecidability of the halting problem (step 1).

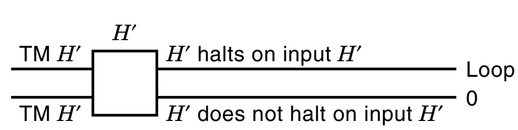

Figure 7. Undecidability of the halting problem (step 3).

PUSHDOWN AUTOMATA

The Model

In this section we use the word *automaton* as a synonym for *acceptor*. Informally, a pushdown automaton consists of an input tape, a pushdown, and a control unit with two pointers, one to the top cell of the input tape (read head) and one to a cell of the pushdown (read–write head); see Fig. 8. One operation on the pushdown is allowed. The automaton can push a new word on top of the pushdown, whereupon the top element will be deleted. The read–write head points to the new top cell of the pushdown. Furthermore, the read head moves one cell from left to right or remains at the old position. It may never move to the left.

A *pushdown automaton* is defined as a structure $M = (\Sigma, \Gamma, Q, \delta, \#, q_0, F)$ where Σ is the *input alphabet;* Γ is the *pushdown alphabet,* including a particular pushdown symbol # called the start symbol; Q is the finite set of *states,* $q_0 \in Q$ is the *initial state,* $F \subseteq Q$ is the set of *final states,* and $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \times Q \times \Gamma^*$ is the *local transition relation.*

A transition $(q, a, z, p, \gamma) \in \delta$ where $a \in \Sigma$, has the following meaning: If M is in state $q \in Q$, reads the input symbol $a \in \Sigma$, and reads the pushdown symbol $z \in \Gamma$, then the automaton will transit to state $p \in Q$, move its read head one cell to the right, and replace the pushdown symbol z by the sequence γ so that the leftmost symbol of γ will be on top of the pushdown. Transitions with $\gamma \in \Gamma^+$ are called *push rules,* because the pushdown store will be enlarged, and transitions with $\gamma = \epsilon$ are called *pop rules,* because the store will be reduced.

A transition $(q, \epsilon, z, p, \gamma) \in \delta$ has the meaning that the read head remains at the same position on the input tape. Therefore the transition can be applied to each configuration where M is in state q and the top symbol of the pushdown is z, independently of the current symbol of the input tape.

Let M be a pushdown automaton as defined above. A *configuration* of M will be described as $z_n \ldots z_1 q a_i \ldots a_m \in \Gamma^* Q \Sigma^*$, where $z_1 \ldots z_n$ is the content of the pushdown and $a_i \ldots a_m$ is the part of the input $a_1 \ldots a_m$ that still can be read. Note that the element at the top of the pushdown is the rightmost symbol z_1, and the element at the bottom of the pushdown is the leftmost symbol z_n in the configuration notation, that is, in reversed order with respect to the transition notation. The *successor configuration* is defined as follows:

1. If $(q, a_i, z_1, p, w_1 \ldots w_k) \in \delta$ is a transition of M, then $z_n \ldots z_2 w_k \ldots w_1 p a_{i+1} \ldots a_m \in \Delta(z_n \ldots z_1 q a_i \ldots a_m)$.
2. If $(q, \epsilon, z_1, p, w_1 \ldots w_k) \in \delta$ is a transition of M, then $z_n \ldots z_2 w_k \ldots w_1 p a_i \ldots a_m \in \Delta(z_n \ldots z_1 q a_i \ldots a_m)$.

Note that in both cases, if $n = 1$ and $k = 0$, then the pushdown will be completely deleted and no further transition is defined.

To finish the definition of the model of pushdown automata we define $\mathsf{in}(u) = \#q_0 u$, $\mathsf{out}(\gamma q \epsilon) = \mathsf{final}(\gamma q \epsilon) = 1$, where $u \in \Sigma^*$, $\gamma \in \Gamma^*$ iff $q \in F$, and otherwise both functions are 0. The pushdown automaton is in a final configuration if it has reached a final state and there is no more input to read. Finally, the *accepted language* of a pushdown automaton M is defined as $L(M) = \{u \in \Sigma^* \mid$ there exists a successful computation of M for input $u\}$.

Considering in detail the definition of pushdown automata, we call a pushdown automaton M *deterministic* if (1) $(q, \epsilon, z, p, \gamma) \in \delta$, then for all $a \in \Sigma$ $(q, a, z, p, \gamma) \notin \delta$, (2) for all $q \in Q, z \in \Gamma$, and $a \in \Sigma \cup \{\epsilon\}$ there exists at most one transition $(q, a, z, p, \gamma) \in \delta$. Otherwise the pushdown automaton is called *nondeterministic.* In contrast to Turing machines and finite automata, the introduction of nondeterminism increases the power of the deterministic pushdown automata model.

Note that a pushdown automaton can be regarded as a restricted Turing machine with two tapes such that each tape has its own read–write head, but the usage of the tapes is restricted. One tape is treated as the input tape, with access restricted to reading from left to right. The access to the second tape, the so-called pushdown, takes place in a LIFO (last in,first out) manner, that is, only the last stored symbol can be read and replaced by a word.

Pushdown automata accept exactly the context-free (type 2) languages. But the following language cannot be accepted by any pushdown automaton: $L = \{0^n 1^n 2^n \mid n > 0\}$. This surprising result can be proved by using a *pumping lemma* for

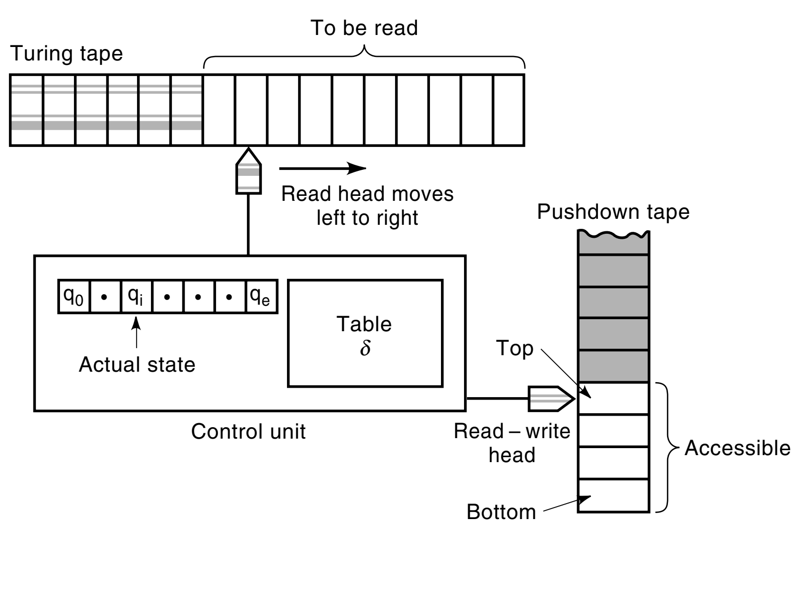

Figure 8. Pushdown acceptor.

context-free languages. The pumping lemma states that if L is a context-free language, then there exists a number k, depending on L, such that each word z in L with length greater than k can be written as $z = uvwxy$ where (1) at least one of v and x is nonempty, (2) the length of vwx is smaller than or equal to k, and (3) uv^nwx^ny is in L for all $n \geq 0$. For more details see Ref. (7).

In contrast to finite automata, there are a lot of undecidable problems concerning pushdown automata and languages they accept. For example, it is not decidable whether two pushdown automata are equivalent (i.e., accept the same language). For more details see Ref. 7.

Examples

We give two examples of languages accepted by pushdown automata.

Example. Let $L = \{a^nb^n \mid n > 0\}$. Then the deterministic pushdown automaton $M = (\Sigma, \Gamma, Q, \delta, \#, q_0, F)$ accepts L, where $\Sigma = \{a, b\}$, $\Gamma = \{b, \#\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_3\}$, and δ consists of the following tuples:

$$(q_0, a, \#, q_1, b\#)$$
$$(q_1, a, b, q_1, bb)$$
$$(q_1, b, b, q_2, \epsilon)$$
$$(q_2, b, b, q_2, \epsilon)$$
$$(q_2, \epsilon, \#, q_3, \#)$$

The automaton M reads all symbols a on the input and pushes for each a an associated symbol b onto the pushdown. After reading all a's, the same number of b's are in the pushdown. Now the automaton compares the b's on the input with the stored b's. In a similar way, a pushdown automaton can be defined to accept the Dyck language D_2. Note that M is deterministic.

Example. Let $L = \{uu^r \mid u = a_1 \ldots a_n, u^r = a_n \ldots a_1, a_i \in \Sigma, 1 \leq i \leq n\}$ and $\Sigma = \{0, 1\}$. A pushdown automaton that accepts L is $M = (\Sigma, \Gamma, Q, \delta, \#, q_0, F)$, where $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \#\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_3\}$, and δ consists of the following tuples:

$$(q_0, 0, \#, q_1, 0\#), \quad (q_0, 1, \#, q_1, 1\#),$$
$$(q_1, 0, 0, q_1, 00), \quad (q_1, 0, 1, q_1, 01),$$
$$(q_1, 1, 0, q_1, 10), \quad (q_1, 1, 1, q_1, 11),$$
$$(q_1, \epsilon, 0, q_2, 0), \quad (q_1, \epsilon, 1, q_2, 1),$$
$$(q_2, 0, 0, q_2, \epsilon), \quad (q_2, 1, 1, q_2, \epsilon),$$
$$(q_2, \epsilon, \#, q_3, \#)$$

The automaton M guesses the middle of the input word and compares the left and the right input part. Since the left part has been stored reversely in the pushdown, a simple comparison with the right part leads to an acceptance or rejection of the input word.

Applications

As an application of the pushdown principle we consider a pocket calculator that uses reverse Polish notation for arithmetic expressions. In such calculators arithmetic expressions have to be entered in postfix notation; for example, $(1 + 2) \times (4 + 5)$ has to be entered as $1\ 2 + 4\ 5 + \times$. The principle of a pushdown automaton can be used in a simple way to implement such a calculator. Numbers are pushed onto the pushdown store until an operator (here $+$) is read. Then the operation is applied by using its arguments from the store. Next the arguments are replaced by the evaluated result (here 1 and 2 are replaced by 3). These actions are repeated until the given expression is completely read. Finally the result can be found on top of the pushdown. In the example, after reading 4 and 5 and replacing it by 9, the application of $\times$ to 3 and 9 leads to the final result 27 stored in the pushdown.

Pushdown automata are of central importance in the area of programming languages and their implementations. If a program is written in a certain programming language, then a so-called parser for that language analyzes the syntactical structure of the program. The parser tries to construct a derivation tree from the input program text. In that case the program is syntactically correct. The syntax of programming languages will be described by context-free grammars of restricted form. One of the most important types of grammars used to define the syntax are the *LR grammars* (7,9,10), which exactly generate the deterministic context-free languages lying properly between the regular languages and the context-free languages. On the automata side the deterministic pushdown automata accept just the deterministic context-free languages and can therefore serve as an implementation basis for those languages. Since pushdown automata can analyze the deterministic context-free languages in a simple and highly efficient way, it is standard to use pushdown automata as the core of a parser. The parser itself is part of a compiler that translates a program into machine-executable code.

FINITE AUTOMATA

The Formal Model of Finite Acceptors

A finite acceptor may be regarded as a pushdown acceptor without a pushdown tape (Fig. 9). It only has its internal finite set of states as a memory. A finite acceptor is defined as a structure $A = (\Sigma, Q, \delta, q_0, F)$ where Σ is the *input alphabet,* Q a finite set of *states,* $q_0 \in Q$ the *initial state,* $F \subseteq Q$ the set of *final states,* and the *local transition* is a relation $\delta \subseteq Q \times \Sigma \times Q$. If this relation is a function $\delta: Q \times \Sigma \rightarrow Q$, then the

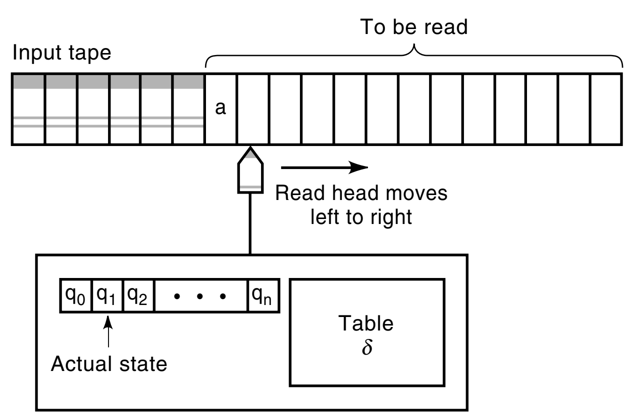

Figure 9. Finite acceptor.

acceptor is called *deterministic,* and otherwise *nondeterministic.* Here we will assume that in the deterministic case δ is a total function, that is, it is defined for all pairs $(q, x) \in Q \times \Sigma$.

The meaning of $(q, x, q') \in \delta$ is that if the automaton A is in state q and reads the input symbol $x \in \Sigma$, then it may transit to state q'. For $(q, x, q') \in \delta$ we also write $q \rightarrow_x q'$, regarding $\rightarrow_x \subseteq Q \times Q$ as a relation on Q for every $x \in \Sigma$. For every word $u = x_1x_2 \ldots x_k \in \Sigma^*$ we define the relation $\rightarrow_u \subseteq Q \times Q$ by letting $q \rightarrow_u q'$ iff there exists a sequence of states $q_1q_2 \ldots q_k$ such that $q \rightarrow_{x_1} q_1 \rightarrow_{x_2} q_2 \rightarrow \cdots \rightarrow_{x_k} q_k = q'$. For the empty word $\epsilon \in \Sigma^*$ we define $q \rightarrow_\epsilon q'$ iff $q = q'$.

The intuitive meaning of the relation $\rightarrow_u$ is that starting in state q and reading the symbol x_1 takes the automaton A to state q_1, and then being in state q_1 and reading x_2 takes A to state q_2, and so on, until x_k takes A from q_{k-1} to $q_k = q'$. We then say that A *accepts* the sequence $u = x_1x_2 \ldots x_k$ iff $q_0 \rightarrow_u q$ and $q \in F$, that is, iff the input sequence u may take A from the initial state to a final state. We define the language L_A accepted by A as $L_A = \{u \in \Sigma^* \mid q_0 \rightarrow_u q \text{ and } q \in F\}$. We immediately conclude that $\epsilon \in L_A$ iff $q_0 \in F$. For any state $q \in Q$ we define the *behavior of* q as the language that A accepts if started in q, that is, $\beta_q = \{u \in \Sigma^* \mid q \rightarrow_u q' \text{ and } q' \in F\}$.

If A is deterministic, there is no choice for the state transition function. Thus if A is in state q and reads the input symbol x, then it deterministically transits to $q' = \delta(q, x)$. An input word $u \in \Sigma^*$ defines a unique q' as the state that is reached from q when the input sequence u is read. We can define this mathematically as an extension δ^* of the function δ to all of Σ^* by induction: $\delta^*(q, \epsilon) = q$, and $\delta^*(q, ux) = \delta(\delta^*(q, u), x)$, where $u \in \Sigma^*$ and $x \in \Sigma$.

A finite acceptor is also intuitively represented by a directed labeled graph with the set Q of states as vertices (nodes) and with a labeled directed edge from q to q' iff $(q, x, q') \in \delta$. So actually δ may be regarded as the set of labeled edges. The initial state q_0 and the final states $q \in F$ are also suitably marked in such a representation (Fig. 10). A finite acceptor reads a word u changing from state q into state q' iff there exists a path in the state graph from q to q' and the label sequence of the path equals u.

In constrast to pushdown automata and similar to Turing machines the models of nondeterministic and deterministic finite acceptor have the same recognition power.

Example. Figure 10 shows the graph of a deterministic finite acceptor with input alphabet $\Sigma = \{a, b\}$ that accepts a sequence $u \in \Sigma^*$ iff u starts with aa and contains a sequence of the word bab.

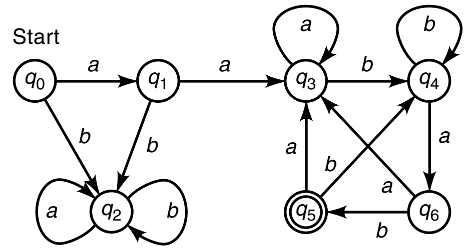

Figure 10. DFA for example.

Regular Languages

We define the class of all languages that can be accepted by a finite acceptor as the class of *recognizable* languages. There is a characterization of the same class of languages by a quite different instrument called *regular expressions.* Regular expressions are similar to the well-known arithmetic expressions, but the meaning of a regular expression is a language and not a number. We want to explain the structure of regular expressions for two reasons. First, they play an important role in a series of software tools for manipulating text (e.g. grep, sed, and shells in Unix) and also for the specification of programming languages (lexical structure). A regular expression defines a pattern that may be searched for in a given text. Second, there is an important theoretical result, given by S. Kleene, that states that the class of recognizable languages is exactly the class of languages that can be defined by regular expressions; see (7).

Let T be an alphabet, that is, a finite set of symbols. We define the set of regular expressions Reg_T recursively as follows:(1) 0 belongs to Reg_T; (2) for all $t \in T$ the symbol t belongs to Reg_T; (3) if α and β are elements of Reg_T, then also the following three expressions belong to Reg_T: $(\alpha + \beta)$, $(\alpha \cdot \beta)$, and (α^*); (4) only expressions that can be formed by rules (1) to (3) in a finite number of steps belong to Reg_T.

Example. Let $T = \{a, b, c\}$. Then the following expressions belong to Reg_T: a, $(a + (a \cdot c))$, (0^*), and $((c^*) + ((b + c) \cdot b))$.

If we agree to the usual precedence rules that the binding of * is stronger than that of $\cdot$, which again is stronger than that of +, then we may omit a few brackets and the above examples may be simplified to a, $a + ac$, 0^*, and $c^* + (b + c)b$. Here we also have omitted the $\cdot$ symbol.

We now can explain how a regular expression α defines a language $L_\alpha \subseteq T^*$. We use a recursive definition based on the recursive structure of the expressions: (1) $L_0 = \emptyset$, the empty language; (2) for all $t \in T$ let $L_t = \{t\}$, the trivial language containing just one word t that in turn consists of the single letter t; (3) if α and β are regular expressions and the languages L_α and L_β are already defined, then $L_{\alpha+\beta} = L_\alpha \cup L_\beta$; $L_{\alpha\beta} = L_\alpha L_\beta$ and $L_{\alpha^*} = (L_\alpha)^*$. So for any regular expression there is defined a unique language that it denotes or specifies. The important property of regular expressions is that a finite expression can define an infinite language.

Example. We take the same alphabet as above and the expression $\alpha = (c + ab)^*$. Then we find that $L_\alpha = \{\epsilon, c, cc, ab, ccc, cab, abc, cccc, ccab, cabc, abcc, abab, \ldots\}$, which is an infinite language.

Sometimes we call L_α the *pattern* specified by the expression α. We call a language $L \subseteq T^*$ *regular* iff there exists a regular expression $\alpha \in \text{Reg}_T$ such that $L = L_\alpha$. Kleene's theorem states that the class of regular languages is exactly the class of recognizable languages. And another theorem from formal language theory states that the class of recognizable languages is exactly the class of languages generated by right-linear grammars (or grammars of Chomsky type 3), which we have defined in the subsection "Hierarchies of Languages and Automata." So we have different tools to specify a regular language.

Because a finite acceptor has only a finite set of states as its memory, the class of languages that are accepted by finite acceptors is rather limited. Turing machines and pushdown machines have also a finite set of states, but their additional storage capabilities (Turing tape, pushdown store) increases their class of accepted languages. When the acceptor reads a word $u \in \Sigma^*$, it traverses its state graph. If the length of u is greater than the number of states of the automaton, then at least one state is visited at least twice. So the memory of the automaton is in the same situation as it was in at the first visit to this state, and thus it cannot distinguish the two situations. As a consequence, if the automaton accepts words of length greater then its number of states, it also must accept all those infinitely many words that are defined by repeating a certain cycle in the state graph any number of times. This is the content of the so-called pumping lemma for finite acceptors or for regular languages. It is closely related to the pumping lemma for context-free languages.

Using the pumping lemma, it can be shown that the following simple language cannot be accepted by a finite acceptor: $L = \{a^n b^n \mid n \in \mathbb{N}\}$. Thus a finite acceptor is not able to recognize even very simple bracket structures. To accept bracket structures we need (at least) a pushdown acceptor.

Minimal Automata

For an engineer it is always important to try to find an optimal (or near-optimal) solution for a given problem. In this section we want to show that for finite automata we are able to construct a minimal automaton with the same behavior. We explain this for finite acceptors, but the ideas carry over also to the finite transducers.

If $A = (\Sigma, Q, \delta, q_0, F)$ is a finite acceptor with behavior $L = L_A \subseteq \Sigma^*$, then we want to know whether A is the only acceptor with this behavior or if there are further acceptors with the same behavior. If two acceptors have the same behavior, we say that they are *equivalent.* It is clear that if we rename every state q to a new symbol, say q', define a new transition function δ' such that $\delta'(q', x) = \delta(q, x)$, and also define the new initial state and set of final states accordingly, then the automaton has not changed substantially. It is said to be *isomorphic* to the original automaton. We expect that the behavior will not change in this case, and that is in fact true.

There are many different but isomorphic automata with the same behavior. They all have the same number of states. This result is not very interesting. But it can be shown that there may exist automata with a smaller number of states accepting the same language or having the same behavior. If this is true, then we may well be interested in finding an automaton with the given behavior and a minimal number of states.

A general result in automata theory says that for any finite acceptor there exists an equivalent acceptor with a minimal number of states. All the minimal acceptors with the same behavior are pairwise isomorphic. So for a given regular language there exists, up to isomorphism, a unique minimal acceptor. This minimal acceptor can be effectively constructed, that is, there is an algorithm that constructs for a given acceptor a minimal equivalent acceptor.

We want to sketch the idea of this procedure. Having defined the behavior of a state of an automaton, we define two states to be equivalent iff they have the same behavior. It can be shown that any two states that have the same behavior can be merged into one state without changing the language of the automaton. The resulting new automaton is called the *quotient automaton* of the given automaton. The quotient automaton is equivalent to the given automaton, and in general has fewer states than the latter, and no two of its states are equivalent. It turns out that the quotient automaton is minimal.

Another result allows for the computation of the minimal number of states for a given regular language without using acceptors explicitly. This is the theorem of Myhill and Nerode; see Ref. (11).

Moore and Mealy Machines

Finite automata that also use an output tape are called finite transducers. They use an extra output alphabet Γ and an *output function* λ. In the literature two types of finite transducers are known as *Moore* and *Mealy machines.* The two types differ in the way the input influences the output. For a Mealy machine the output symbol depends on the actual state and on the input symbol, so we have $\lambda : Q \times \Sigma \to \Gamma$. For the Moore machine λ only depends on the actual state, so $\lambda : Q \to \Gamma$. Transducers can also be represented by directed graphs, like acceptors, but now the output function λ must also be included. For a Moore machine we simply assign the output symbol $\lambda(q)$ to the node q, and for a Mealy machine an edge $q \to_x q'$ will additionally be labeled with the output symbol $y = \lambda(q, x)$, which will be denoted as $q \to_{x|y} q'$ (Fig. 11). In this section we will only regard the case of the Mealy machine.

Given a Mealy machine $M = (\Sigma, Q, \Gamma, \delta, \lambda, q_0, F)$, we can extend the output function λ to input sequences by a recursive definition: (1) $\lambda^*(q, \epsilon) = \epsilon$; (2) $\lambda^*(q, ux) = \lambda^*(q, u)\, \lambda(\delta^*(q, u), x)$ for $u \in \Sigma^*$ and $x \in \Sigma$. So, given state $q \in Q$, the empty input gives an empty output, and if the input sequence u produces the output sequence $\lambda^*(q, u)$ and u transforms q to $q' = \delta^*(q, u)$ (i.e., $q \to_u q'$), then the output symbol $\lambda(q', x)$ is appended to the sequence $\lambda^*(q, u)$ that has been produced so far. With every state $q \in Q$ there is associated the function $\beta_q : \Sigma^* \to \Gamma^*$ defined by $\beta_q(u) = \lambda^*(q, u)$. The function β_q is called the *behavior* of q. The input–output behavior of the machine is then defined as the behavior of the initial state q_0. Note that λ^* is just concatenating the outputs along a path. A function $f : \Sigma^* \to \Gamma^*$ that is the behavior of a Mealy machine is called a *sequential function.*

Applications

The theory of finite automata is a very rich theory with many important and interesting results. We only have given a short summary of a few of these results. We could not even give all

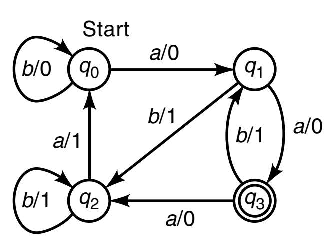

Figure 11. Mealy automaton.

the necessary background to explain, for example, the theory of the decomposition of finite automata (13) or the theory of stochastic automata (14).

Moore and Mealy machines are very important abstract models for synchronous switching circuits. A switching circuit consists of a number of binary storage elements (flip–flops) and function elements that realize Boolean functions. Thus a state is the (stable) state of all the flip–flops and defines a vector or list $(s_1, \ldots, s_k) \in \mathbf{2}^k$ of binary values. Also, the input of such a network is a binary vector, namely, the list of all the binary values $(i_1, \ldots, i_m) \in \mathbf{2}^m$ applied to the m input connectors, and the output is a vector $(o_1, \ldots, o_p) \in \mathbf{2}^p$ of the p output connectors, see Fig. 12.

If a technical problem is given as a description of a sequential input–output function for abstract input and output sets Σ and Γ, then we may start by constructing a Mealy machine M that has the given function as its behavior. In a second step the state set Q and the input and output alphabets Σ and Γ must be represented (encoded) as suitable lists (vectors) of binary values. After these encoding functions have been defined, the state transition function δ and the output function λ have to be realized as Boolean (logical) functions. The structure of these functions defines the combinatorial part of the switching network. The state vector $(s_1, \ldots, s_k) \in \mathbf{2}^k$ is represented by a set of flip–flop elements. Of course, the logical functions depend on the choice of these elements. But it is a very interesting result of the theory of the realization of automata that the choice of the encoding functions also has a significant influence on the complexity and structure of the combinatorial part of the switching network. Hartmanis and Stearns have developed a rich theory for the realization of finite machines (12).

An important application of finite acceptors is the construction of a compiler for a programming language. In this case the input text is a computer program in a defined programming language. In the first stage of processing, the compiler tries to split the input text into subsequences (lexemes) that fall into a number of different syntactic classes or patterns such as identifiers or numbers. These (linear) patterns are specified by regular expressions or by right-linear grammars and thus may be recognized and classified by a set of finite acceptors, one for each different pattern. Such a system of finite acceptors is then simulated by an algorithm,which is known as a scanner. The scanner performs the lexical analysis of the input text.

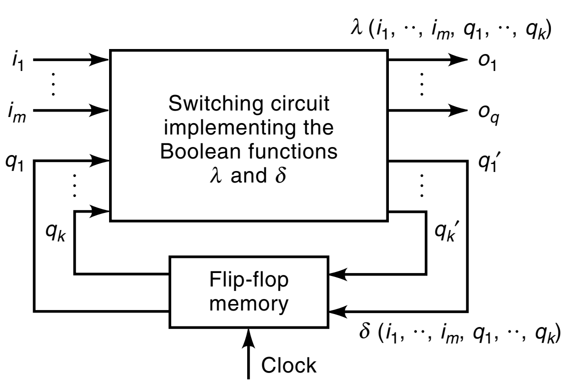

Figure 12. Huffman model of sequential switching network.

Finite transducers are used to model information-processing devices, which then may be realized by electronic circuits, as we already have discussed for switching networks.

As an example of a recent application of the theory we want to mention an algorithm for the compression of grayscale pictures that is based on finite automata that have real numbers as edge labels in their graphical representation (15). This special form is called a weighted finite automaton. A weighted finite automaton with state set $Q = \{q_1, q_2, \ldots, q_n\}$ uses a set of $n \times n$ matrices with real entries, one for each input symbol $x \in \Sigma = \{0, 1, 2, 3\}$. The input symbols are chosen so that every word $u \in \Sigma^*$ defines a subsquare $f(u)$ of the unit square I in the real plane $\mathbb{R}^2$. This mapping is known as the quadtree mapping (Fig. 13). The squares that are assigned to the words $u \in \Sigma^k$ of fixed length k all have length 2^{-k} and define a partition of the unit square. If we assign a gray value to each of these 4^k squares, then we have an image of finite resolution $2^k \times 2^k$ pixels. So we may define this image by assigning a gray value to each of the words $u \in \Sigma^k$. Let M_x be the $n \times n$ matrix defined for the symbol $x \in \Sigma$, $\xi \in \mathbb{R}^n$ an initial row vector, and $\eta \in \mathbb{R}^n$ a column vector of weights for the final states.

For a sequence $u \in \Sigma^*$ we can define the product matrix M_u recursively: (1) $M_\epsilon = E_n$, the $n \times n$ unit matrix; (2) $M_{ux} = M_u M_x$ for $u \in \Sigma^*$ and $x \in \Sigma$. Mathematically this is a matrix representation of the free semigroup Σ^*. Now we can use this representation to assign a real number to any $u \in \Sigma^*$ by defining $\phi(u) = \xi M_u \eta$, where $\phi: \Sigma^* \to \mathbb{R}$. Such a function ϕ is called *average-preserving* iff for all $u \in \Sigma^*$ it holds that $4\phi(u) = \sum_{x\in\Sigma} \phi(ux)$. This property guarantees that each square $f(u)$ has a gray value $\phi(u)$ that is the average of the gray values of its four subsquares. If for a weighted automa-

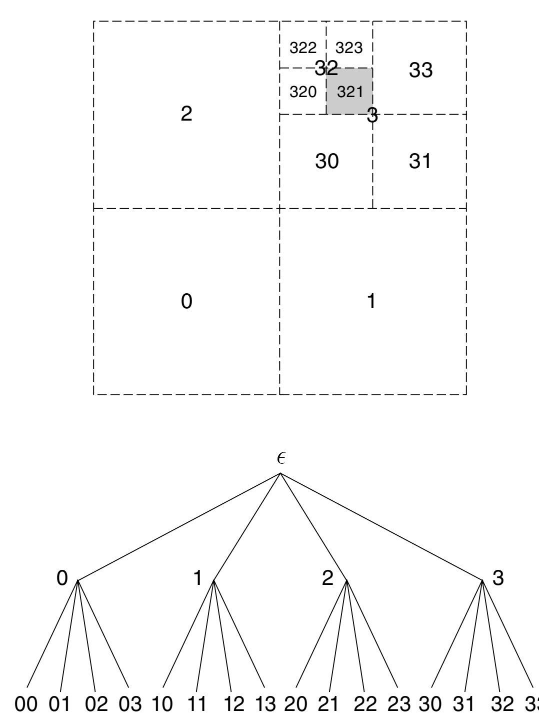

Figure 13. Quadtree mapping from words to subsquares of the unit square.

ton the function ϕ is average-preserving, then it defines an image of arbitrarily high resolution.

For images of finite resolution the algorithm of Culik and Kari finds a weighted automaton that approximates the image with a given measure of distortion (15).

BIBLIOGRAPHY

1. A. M. Turing, On computable numbers, with an application to the Entscheidungsproblem, *Proc. London Math. Soc.,* Series **2** (42): 230–265, 1936–1937.
2. M. Davis, Hilbert's tenth problem is unsolvable, *Amemm* **80** (3): 233–269, 1973.
3. G. Rozenberg and A. Salomaa, *Handbook of Formal Languages,* Vol. 1, 2, 3, New York: Springer-Verlag, 1997.
4. J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science,* Vol. A, B, C, Amsterdam: Elsevier, 1990.
5. M. R. Garey and D. S. Johnson, *Computers and Intractability,* New York: Freeman, 1979.
6. R. E. Kalman, P. L. Falb, and M. A. Arbib, *Topics in Mathematical System Theory,* New York: McGraw-Hill, 1969.
7. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation,* Reading, MA: Addison-Wesley, 1979.
8. M. L. Minsky, *Computation: Finite and Infinite Machines,* Englewood Cliffs, NJ: Prentice-Hall, 1967.
9. A. Salomaa, *Formal Languages,* New York: Academic Press, 1973.
10. J. G. Brookshear, *Formal Languages,Automata, and Complexity,* Menlo Park, CA: Benjamin Cummings, 1989.
11. D. I. A. Cohen, *Introduction to Computer Theory,* New York: Wiley, 1986.
12. J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines.* Englewood Cliffs, NJ: Prentice-Hall, 1966.
13. S. Eilenberg, *Automata, Languages, and Machines,* New York: Academic Press, 1976.
14. A. Paz, *Introduction to Probablistic Automata.* New York: Academic Press, 1971.
15. K. Culik II and J. Kari, Inference algorithms for WFA and image compression, in Y. Fisher (ed.), *Fractal Image Compression,* New York: Springer-Verlag, 1995.

WOLFGANG GOLUBSKI
WOLFGANG MERZENICH
University of Siegen

AUTOMATED GUIDEWAY TRANSIT. See AUTOMATIC GUIDED VEHICLES.