5.2: Basic Concepts of Automata Theory

R. J. Nelson

Case Institute of Technology, Cleveland, O.

I WAS INVITED to present a survey of the theory of automata. Since there are already several surveys, especially the very competent ones of Chomsky[10], McNaughton[37], and Rogers[54], all of which deal with various aspects of the theory of automata and more, I shall confine myself to an exposition of what might loosely be called a "philosophy of automata". On the way, however, I hope to touch upon certain developments of the subject which have come about since McNaughton's 1961 paper. Readers who feel they are up on the theory may wish to omit sections 2 and 3.

What the theory is about.

An understanding of what automata theory is about presupposes some familiarity with logic and recursive function theory. We are interested in functions of non-negative integers into the non-negative integers which can be calculated by an algorithm. These functions are any of the list,

$$\begin{aligned} &\text{(a) } S(x) = x+1, \\ &\text{(b) } U_i(x_1, \ldots, x_n) = x_i, \\ &\text{(c) } x+y, \\ &\text{(d) } x \dot{-} y, \\ &\text{(e) } xy, \end{aligned} \qquad (1)$$

or are obtainable from (a) -- (e) by application of the operations of composition and minimalization.

Three items in the above formulation may be unfamiliar. (b) is the generalized identity function, U_i, whose value for an assignment of value to the n variables is the value assigned to x_i. (d) is proper subtraction, and has the value x - y (ordinary subtraction) if $x \geq y$ and the value 0 otherwise. Minimalization is an operation on an equation $f(y,x_1, \ldots, x_n) = 0$. The value of the operation is the smallest non-negative integer y satisfying the equation; if there is no such y the operation is undefined. The function f in the equation must be one already obtained by the foregoing process.

A function of n variables obtained by the indicated procedures, if it is defined for some values of the variables and not others, is partial recursive; if it is defined for all values, then it is recursive. A sub-set of the recursive

functions is the set of primitive recursive functions. Roughly speaking these functions are defined by a "recursive process" in the sense that the function value f(0) is given outright and f(y+1) is defined in terms of both f(y) and y. This idea is applicable to functions of y with parameters $x_1, \ldots, x_n$ (Davis[13]).

Example. y+y is recursive by (c) above. By composition and (d) $(y + y) \dot{-} x$ and $x \dot{-} (y + y)$ are recursive. Again by composition,

$$[(y+y) \dot{-} x] + [x \dot{-} (y+y)] \qquad (2)$$

is recursive. But (2) equals $|2y - x|$, which is therefore recursive. Finally by minimalization, $\frac{x}{2}$ is the least integer y which satisfies $|2y - x| = 0$. . The function $\frac{x}{2}$ is not defined for odd integers x (there is no least y satisfying the equation in such cases), hence it is partial recursive.

A recursive set of integers is one which has associated to it a two-valued recursive function such that if an integer is in the set the function has the value 0 or "yes" and otherwise 1 or "no". For example, the set of integers greater than or equal to 10 is recursive since the associated function $1 \dot{-} (1 \dot{-} (10 \dot{-} x))$ is recursive.

The set of values of a recursive function; i.e., its range, is a recursively enumerable set. A set is recursive if and only if both it and its complement are recursively enumerable.

Now a word of interpretation is called for. Recursive function is a concept meant to make completely exact the vague notion, used by mathematicians and computer people, of an "effectively calculable" function, or of a function calculable by a systematic procedure or algorithm -- in other words, by a simple clerk. Similarly, recursive set is a precise notion corresponding to the intuitive idea of mechanical question answering. "Is there an algorithm for deciding whether n is prime?" now becomes the precise question "Is the set of primes recursive?" Finally, recursively enumerable set clears up the vague idea of an "effective enumeration" -- generating a (possibly infinite) sequence of integers by a "mechanical" or "rote" process. We now may talk instead about the range of a recursive function.

These three key ideas can all be extended to the domain of non-numerical problems. Suppose T is any transformation from one set into another, say from the set of English sentences into the set of Russian sentences. If you like, assume that T preserves "meaning", whatever that is. Now if an effective one-one correspondence can be set up from English sentences to numbers, and another from Russian sentences to numbers; and if the function f_T which associates n to $f_T(n)$ when and only when the English sentence corresponding uniquely to n is translated by the Russian sentence uniquely corresponding to $f_T(n)$ is partial recursive, then so is T. In such wise we may talk meaningfully about recursive transformations or functions with respect to arbitrary sets. Of course, by analogous devices we extend the concept recursive set and recursively enumerable set to arbitrary collections -- for the most part to sets of symbols, words, sentences, languages, etc. For instance, the set of theorems of any of the usual systems of logic is recursively enumerable -- and this fact justifies programing proof procedures on digital computers. If a formula is a theorem, a proof can be found by an algorithm. Again, if a set of occurrences of the letter "A" is recursive -- no matter how poorly registered, degraded, or from

how many type fonts -- an occurrence can be recognized or not as an instance of "A" by an algorithm.

Automata theory studies models of machines of various kinds which can compute recursive functions, and which can either recognize or generate the elements of recursive or recursively enumerable sets. This is not yet a definition, but is, I claim, a good description.

The machine models we are looking for are systems of rules for operating on symbols. We have at hand, suppose, a non-empty finite set of symbols S called the alphabet. By S* we understand all of the finite sequences of expressions made up out of the symbols of S. These sequences are called words; if x is a sequence on S, S (i.e., if $x \in S^*$) and $y \in S^*$, then $z = xy$ is also a sequence on S, i.e., $z \in S^*$. The act of tacking y onto x is concatenation. S* is a denumerably infinite set. S* includes the null word Λ which has the property $\Lambda x = x\Lambda = x$, for any $x \in S^*$.

Let P and R be variables over S* and let the symbols g, g' be constants denoting fixed words of S*. A rule is any expression of the form

$$PgR \rightarrow Pg'R, \tag{3}$$

which is to be interpreted as meaning: given g as part of an expression PgR of S*, to rewrite g by g' yielding Pg'R.

Using the standard terminology of the ALGOL 60 reference language[47] as the alphabet in our sense,

P basic symbol R → P letter R

and

P basic symbol R → P digit R

are rules. It is easy to see that our rules of the form (3) are recursive, which simply means that to rewrite the symbols g by g' can be done by an algorithm. Technically the assertion means that the set of ordered pairs (α, β) such that there are words P and R for which $\alpha = PgR$ and $\beta = Pg'R$ are true, is a recursive set.

We say that β is a consequence of α by the rule $\alpha \rightarrow \beta$. We also say informally that β follows from α.

Ordinarily we are interested in the cases where we have some given word $w \in S^*$ which we operate on by using rules. Such a word is an initial word or axiom. In

the ALGOL reference language, "basic symbol" is an initial word or axiom.

Consider a much simpler example with S = {0, 1}, S^* = {Λ, 0, 1, 00, 01, 10, 11, 000, ...} and an initial word 1010111. The set of rules for the example is

(a) P 10 R → P 01 R

(b) P 11 R → P 1 R

(c) P 00 R → P 0 R

By applying these rules to the axiom we generate a sequence $w = w_1, w_2, \ldots, w_8$ in which w_{i+1} follows from w_i by some rule (i = 1, ..., 7). Such a sequence is derivation, and the last word generated, w_8, is called a result or theorem.

Let us see how the rules in fact do generate a sequence. We write the sequence as a list and to the right we annotate the entry.

(1)	1010111	initial word;
(2)	0110111	using 1 as the left side of rule (a) with g = 10, g' = 01, P = Λ and R = 10111;
(3)	010111	rule (b);
(4)	001111	rule (a);
(5)	00111	rule (b);
(6)	0111	rule (c);
(7)	011	rule (b);
(8)	01	rule (b).

If a result has no consequence, then it is terminal. The above result, 8, is terminal since no rule applies to the word 01.

Let us now attempt to design a system (with the same alphabet as above) of rules to operate on a word w and produce a result w' as follows. Let n be the number of occurrences of 1 in w; if n is even, the result w' is to have n/2 occurrences of 1 in it; if n is odd, w' is to have $\frac{n+1}{2}$ occurrences of 1 in it. Any distribution of 1's in w' is permitted.

Anyone who tries to fashion a set of rules for performing the desired task will find that: (1) - the rules must effect a scan across the initial word w from left to right (or oppositely); and (2) - special markers, distinguishable from the symbols of S, must be available to keep count of the oddness or evenness of the number of occurrences of 1's scanned up to any step in the derivation. We call these special markers auxiliary symbols.

Continuing the example, we need besides the alphabet S = {0, 1, #} where we use # as an end marker, an auxiliary alphabet Q = $\{q_0, q_1\}$. We set A = S U Q, and in our rules let P, R range over A^*. As before x, y, w, etc., will be words on S - { x } alone. q_0w# is the initial word or our system. In this and later cases where the initial word is of this form w will be called the input. The rules for the system are the following six.

(a) P q_0 0 R → P 0 q_0 R

(b) P q_0 1 R → P 1 q_1 R

(c) P q_1 0 R → P 0 q_1 R

(d) P q_1 1 R → P 0 q_0 R

(e) P q_0 # R → P # R

(f) P q_1 # R → P # R

Suppose the input is 1011101. When the initial word is

(1) q_0 1011101#

Rule b applies, yielding

(2) 1q_1 011101#

If one applies rules (c), (d), (b), (d), (a), (b), and (e) in order, he will obtain

.
.
.

(9) 1001001#

The part of this terminal result up to # is an output.

Returning to the example from ALGOL let us further illustrate some of the above described ideas. We shall see that an interesting fragment of ALGOL **numbers** is again a system of recursive rules with an alphabet and auxiliaries, an initial word, derivations, and results in the foregoing sense. Here we let our pure alphabet be

$$S = \{ ., +, -, 0, 1, \ldots, 9, 10 \}.$$

The auxiliary alphabet is

$$Q = \{ D, UI, DF, EP, DN, UN, N \}.$$

There is a single initial word which is the auxiliary N. In this system to shorten the list of rules we will use the following convention.

If PgR → Pg'R and PgR → Pg''R are two rules, we express them as one by (a) suppressing use of the syntactical variables P and R with the understanding that g may be replaced by g' (or g'') in any context, and (b) by indicating by use of the expression g'|g'' that g' or g'' may replace g. Thus

P U I R → P D R

P U I R → P U I D R

is to be written simply

(1)" UI → D|UID.

The other rules for our ALGOL fragment are the following seven:

(2)" I → UI|+UI|-UI

(3)" DF → .UI

(4)" EP → $_{10}$I

(5)" DN → UI|DF|UIDF

(6)" UN → DN|EP|DNEP

(7)" N → UN|+UN|-UN

(8)" D → 0|1|2|3|4|5|6|7|8|9

Those familiar with ALGOL will see that this is a system of rules for generating numbers. Although the "semantics" of the auxiliary symbols are irrelevant, it may be helpful for comprehension to give an interpretation.

D = decimal, UI = unsigned integer, DF = decimal fraction,
EP = exponent part, DN = decimal number, UN = unsigned number,
and N = number.

We may <u>derive</u> the real number $-2.46_{10}67$ as follows:

(1)	N	the initial word;
(2)	-UN	from Rule 7.";
(3)	-DNEP	from Rule 6.";
(4)	-UIDFEP	from Rule 5.";
	.	
	.	
	.	
(17)	$-2.46_{10}67$	using

rules 1", 8", 3", 1", 8", 1", 8", 4", 2", 1", 8", 1", and 8" in that order.

Let us emphasize that 1" -- 8" are abbreviations of rules in our fundamental sense and hence that we have a system of rules satisfying the same kind of properties

as the pulse divider.

Bringing together the various parts of our discussion thus far, we now assert that any finite set of rules $PgR \rightarrow Pg'R$ with a single axiom and a finite non-empty alphabet with at least two auxiliaries is an automaton. This characterization seems to be fairly adequate since it includes Turing machines, finite automata, pushdown automata, phrase structure grammars, idealized nerve networks, linear bounded automata, sequential machines, etc. If we extend the notion of automata to systems which are equivalent in the sense that they yield the same terminal results from the same inputs, then computer programs are automata since they can in effect be treated as formal systems (Shepherdson and Sturgis[58]), (Nelson[48]). Since it expresses only a sufficient condition the above characterization includes too much -- most familiar systems of logic are automata. On the other hand, stating sufficiency leaves the door open to switching circuits, probabilistic automata, perceptrons, analog computers, control systems, and learning machines. However, we will have enough on our hands just to talk about systems satisfying the condition.

An automaton may have any number of physical embodiments. A system of rules on paper is an automaton as is a solid state sequential circuit. Also, being an automaton is relative to a point of view and does not preclude being something else from another point of view. Thus, as McNaughton points out, a clock is an automaton, albeit a trivial one, but it is also an energy system.

The study of automata is co-extensive with much of logic and hence is a part of mathematics. As such it shares the strengths and weaknesses of mathematics. One can establish the theory on a rigorous basis and explore a realm of possibilities not immediately obvious to the programmer or designer and yet at the same time admit to idealization and the need for adequacy checks against the real world of problems, which occasion the theory in the first place.

There appear to be three main varieties of automata: acceptors, generators, and transducers or sequential machines.

Let S be any alphabet including #, Q any set of auxiliaries, $A = S \cup Q$, $S \cap Q = \phi$ (ϕ designates the null set), and let symbols P, R, x, y, etc. be used as before. An acceptor is an automaton having a single axiom qw with or without end markers, and such that every terminal result is of the form q. If qw is the axiom, then an acceptor accepts w if and only if there exists a derivation in the acceptor yielding a terminal result of the form q. A set of words is definable if there exists an acceptor that accepts it.

A generator is an automaton having a single axiom of the form q and such that every terminal result is a word on symbols of $S - \{\#\}$ only. A terminal result of a generator is called a sentence and the set of all such sentences of a generator is a language. Generators are also called generative grammars when the rules are constructed in such a way that the derivation of a sentence reveals its grammatical structure. The study of languages from this point of view is part of mathematical linguistics; it will be obvious from the ensuing discussion in section 2 that linguistics and automata theory have much in common. The rules defining the ALGOL fragment above constitute a generative grammar.

A transducer or sequential machine is an automaton having a single axiom of the form qw# or q#w#, and such that every terminal result is a word on the symbols of $S - \{\#\}$ only. The pulse divider previously discussed is a finite transducer. In qw#, w is an input and the terminal result, say w', is an output. The set of all

ordered pairs (w, w') of this kind is the behavior of the transducer, and if this set is a function, the transducer represents it.

From now on we suppose each automaton to include a distinguished auxiliary q_0 -- the initial state -- and we suppose every q in an axiom is q_0.

Automata Behavior

Although I claim that it is essentially correct to describe "automata" as I have in terms of rules, it will not be particularly convenient or even illuminating for the remainder of this discussion to persist in the formalities in all cases. Suffice it to say that every automaton we shall discuss can be completely and correctly defined as a system of rules.

The most powerful automaton is the Turing machine*. As Turing machines are very well known and have been described in many papers and books we will not redescribe them here.

A Turing machine is a transducer in our sense of the word, and it computes a function of the non-negative integers under the following circumstances. If the argument (or n-tuple of arguments) is coded in the machine's alphabet on tape, and if the machine then derives a terminal result representing the function value, then we say it computes the value of the function from the argument. A function for which there exists a Turing machine which computes the value for each integer (or n-tuple of integers) is computable.

Proposition 1. A function is (partial) computable if and only if it is (partial) recursive.

On informal grounds proposition 1 means that Turing transducers can handle any function or transformation or translation for which there is an algorithm. Conversely, if we accept the thesis** that all effectively calculable functions are recursive, a procedure is an algorithm only if it can be done on a Turing machine.

By altering the defining rules of a Turing machine we may obtain a Turing acceptor and a Turing generator. Like the transducer, a Turing acceptor can move back and forth reading and writing on an infinite tape. It should be expected that such a device could accept very weakly structured sets. Indeed,

Proposition 2. A set is accepted by a Turing acceptor if and only if it is recursively enumerable.

Similarly,

Proposition 3. A set is generated by a Turing generator if and only if it is recursively enumerable.

Propositions 2 and 3 have two consequences, among others, which are of immediate interest to computer scientists. The first one is that the theorems of any formal theory can be generated by a Turing machine, since such theorems are known to comprise a recursively enumerable set. Assuming that a digital computer is a Turing machine (we shall question this assumption later), this fact justifies non-heuristic theorem proving programming studies. The second one is that natural languages such as English or Russian or artificial languages such as ALGOL or FORTRAN are either generable by or accepted by Turing automata. As we shall see, the same is true of considerably weaker automata.

Turing himself showed the existence of one of his machines which can compute any recursive function. Shannon[57] has shown that such a universal machine can do with two symbols, {0, 1}, at the cost of increasing greatly the number of states -- i.e., auxiliaries; or it can do with two states if there is a sufficiently large alphabet of symbols. Watanabe[62] has constructed a universal machine having a state-symbol product of only 30!

The weakest kind of automaton is a finite state acceptor (f.s.a.)***. An f.s.a. has rules of the forms

$$\begin{aligned} &\text{(a)} \quad P\ q\ s\ R \rightarrow P\ q'\ R \\ &\text{(b)} \quad P\ q\ \#\ R \rightarrow P\ q'\ R \end{aligned} \tag{4}$$

subject to sundry provisos, the main one being that each pair $(q, s) \in Q \times S$**** occurs exactly once on the left side of a rule of type (a). If Q has n elements and S has m, then there will be m x n rules of the (a) form. Given a subset $Q' \subseteq Q$, called the set of final states, a word $w \in S^*$ is accepted if and only if there is a derivation from $q_0\ w\#$ with terminal result q', $q' \in Q'$. The auxiliary q_0 is called the initial state.

Inspection of the rules making up an f.s.a. shows that we may picture such automata as Turing acceptors which read tape from left to right only and which do not write. Still more suggestively, an f.s.a. is a "black box" with input, capable of receiving an input symbol by symbol. If the automaton goes into a final state of Q' after injection of a word, a green light flashes.

As an example, consider the f.s.a. with $S = \{0, 1\}$, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $Q' = \{q_4\}$ and the rules

(a) $P\ q_0\ 0\ R \rightarrow P\ q_0\ R$

(b) $P\ q_0\ 1\ R \rightarrow P\ q_1\ R$

(c) $P\ q_1\ 0\ R \rightarrow P\ q_0\ R$

(d) $P\ q_1\ 1\ R \rightarrow P\ q_1\ R$

(e) $P\ q_2\ 0\ R \rightarrow P\ q_0\ R$

(f) $P\ q_2\ 1\ R \rightarrow P\ q_3\ R$

(g) $P\ q_3\ 0\ R \rightarrow P\ q_4\ R$

(h) $P\ q_3\ 1\ R \rightarrow P\ q_0\ R$

(i) $P\ q_4\ 0\ R \rightarrow P\ q_4\ R$

(j) $P\ q_4\ 1\ R \rightarrow P\ q_4\ R$

(k) $P\ q_4\ \#\ R \rightarrow P\ q_4\ R$

This f.s.a. accepts words having at least one occurrence of three consecutive 1's in it. For example, the word q_4 is a terminal result derivable from the axiom q_0 1011101#.

The sets definable by f.s.a. are readily characterized in terms of concatenation. Using our standard alphabet S, if A and B are subsets of S^*, the complex product, written A·B, is the set of all words xy such that $x \in A$ and $y \in B$. Also let us write A^2 for A·A, A^3 for A·A·A, etc. The iterate of A on A or the star of A, written A^*, is the infinite union of all finite powers of A:

$$A^* = \bigcup_{i=0}^{\infty} A^i , \qquad A^0 = \{ \Lambda \}$$

Incidentally, this definition shows why we used the notation S^* for all of the words on S in the first place.

Let us now define the notion of regular set recursively as follows. Any finite set of words is regular. If A and B are regular then A·B is regular, A U B (the union of A and B) is regular, and A^* is regular.

As an example, with $S = \{0, 1\}$, it is fairly easy to see that the set of words containing an even number (including zero) or 1's is regular. If we let "0" itself designate the set consisting of 0 only and "1" designate the set consisting of 1 only, then the desired regular set is denoted by

$$(0^* \cdot 1 \cdot 0^* \cdot 1)^* .$$

In more or less ordinary English, this says a word consisting of any number (including zero) or 0's followed by a 1 followed by any number of 0's (including zero) followed by 1 is in the set; and moreover that the result of repeating the process just described any number of times yields words with an even number of 1's.

We are now in the position to state a fundamental result in the theory of finite acceptors (Kleene [32]).

Proposition 4. U is accepted by some f.s.a. if and only if U is a regular set. The definable sets are precisely the regular ones.

The complement of the set of words having an even number of 1's is given by the expression

$$(0^* \cdot 1 \cdot 0^*)^*$$

and contains words with an odd number of ones. It is easy to show that every f.s.a. not only accepts words of U but rejects words of $\bar{U}$, the complement of U. Hence, by redesignating the final states of the given f.s.a. we can obtain another f.s.a. which accepts $\bar{U}$. Hence by proposition 4 and the fact that union of regular sets are regular, we have

Proposition 5. The family of all regular sets is a boolean algebra. (Rabin and Scott[52]).

Since an f.s.a. is a restricted Turing machine, it follows by proposition 2 that regular sets and their complements are both recursively enumerable. Hence, regular sets are recursive.

It is now fair to ask, so what? I think careful understanding of even the simplest automaton is valuable in itself. However, the theory about f.s.a. is clearly relevant to pattern recognition. Suppose we have a character set of n (say 64) characters each of which can be registered in a multitude of deformed, mal-positioned, or degraded ways. We can build a recognizer (using f.s.a.'s only) if and only if each set of possible occurrences of a character (distorted badly but within limits) is regular, and the sets are disjoint. Each word representing a scanned character is "fed into" (is an initial word of) several f.s.a. simultaneously, where the f.s.a. correspond uniquely to the regular sets. It would, of course, be folly to attempt character recognition in this way if the sets of character tokens were irregular. Keller[31] has made an analysis of certain pattern recognition schemes, such as the perception making use of the f.s.a. and related ideas. The theory also provides a very primitive model of what it means to "understand" a sentence.

A remarkable result discovered by Rabin and Scott[52] concerns a class of acceptors which are not constrained in each step of the derivation to assume a unique next state. They allow of the rules**** defining an acceptor that there be at least one production rule with left side P q s R for each pair $(q, s) \in Q \times S$. In other words, a situation such as

$$\begin{aligned} P\ q_0\ s_1\ R &\rightarrow P\ q_1\ R \\ P\ q_0\ s_1\ R &\rightarrow P\ q_2\ R \end{aligned} \tag{5}$$

is permitted. A word $w = \ldots q_0 s_1 \ldots$ has possibly two (or more) consequences in a derivation. Such systems of rules are said to be polygenic, while the systems for ordinary f.s.a. are monogenic since each word in a derivation has at most one immediate consequence. An acceptor like an f.s.a. except for having polygenic rules is called a non-deterministic finite state acceptor (n.f.s.a.). A word w is accepted by an n.f.s.a. if there exists a derivation yielding an auxiliary terminal result in Q'. It would not seem unreasonable to believe that an n.f.s.a. could accept more than a regular set -- that is other than regular sets -- sets having weaker defining properties. But this is not the case.

Proposition 6. Every set accepted by an n.f.s.a. is accepted by some f.s.a. (Rabin and Scott[52]).

Further results about simple acceptors in zig-zag tapes, two-tapes, etc., are described in Rabin and Scott[52].

Probabilistic Automata (objects not satisfying the sufficiency condition for automata in our sense) may be looked upon as systems like the f.s.a. except that each pair $(q, s) \in Q \times S$ leads to a new state q' with a certain transition probability p' (q, s). In general these automata can accept sets not within the range of acceptability of the f.s.a. However, there are two somewhat surprising exceptions which should be briefly discussed.

If Q' is the set of final states of a probabilistic acceptor P, let p(x) be the probability that the automaton will go into a state of Q' from the initial state q_0 upon injection of an input word x. x is accepted by P if p(x) is greater than some previously given real number λ, $0 \leq \lambda < 1$. For emphasis we might say that in such a case x is λ-accepted.

Proposition 7. If $| p(x) - \lambda |$ is bounded from below, namely, if there is a real positive δ satisfying

$$\delta \leq | p(x) - \lambda | \text{ , for every } x \in S^*$$

then P accepts only regular sets (Rabin[51]). Secondly, if P satisfies the above property and if in addition $p'(q, s) > 0$ for every $q, q' \in Q$ and $s \in S$ then the set of accepted words is even more constrained than the regular sets. Such an automaton can "do" only less than an f.s.a.!

It appears that these results should have important implications for the design of reliable sequential machines out of unreliable components. However, we will not attempt to discuss the matter further here.

A finite state generator or grammar is essentially an acceptor "turned around". All of the rules are of the form

$$P\ q\ R \rightarrow P\ q'\ s\ R \tag{6}$$

or

$$P\ q\ R \rightarrow P\ s\ R$$

subject to sundry provisos. In general all generators will be non-deterministic. It is known (Bar-Hillel and Shamir[3]) that the sets or words generated by such generators are regular. Hence as in the case of Turing automata, finite state acceptors and generators are equivalent in the sense that they are both associated with the same family of sets.

Proposition 8. A set is regular if and only if it is the set of words generated by a finite state grammar.

As noted before, the terminal results of a generator are called sentences, and the set thereof a language. Also, it is customary in dealing with grammars to term the auxiliary symbols Q (called "states" in acceptors and transducors) non-terminals, and the symbols S terminals. In application to linguistics the terminals are usually words of the natural language under study. The symbols N (noun), NP (noun phrase), V (verb), etc. are generally used as non-terminals in order to suggest grammatical concepts, but for the sake of uniformity of the treatment we shall continue to use q symbols.

A sequential machine or finite transducer (f.t.) is a device similar to an f.s.a. but with output. It computes in strictly real time in the sense that each output from an input is computed instantaneously, and an output sequence is formed exactly when an input sequence is introduced. Equivalently, a finite transducer is a Turing machine with a one-way (left to right) tape. In formal terms, an f.t. is

characterized by a finite set of rules of the form

$$P\ q\ s\ R \rightarrow P\ O\ q'\ R \qquad (7)$$

$$P\ q\ \#\ R \rightarrow P\ \#\ R$$

where there is at most one rule for each $(q, s) \in Q \times S$ (determinism). "O" is an output symbol of an alphabet O which may or may not be the same as S: Hence $A = S \cup Q \cup Q$; and we require $(S \cup O) \cap Q = \phi$.

We wish to characterize the behavior of such automata. This objective is somewhat more practical than the like one for Turing machines, since sequential machines are models of ordinary computer sequential circuits; e.g., the pulse divider.

It is clear that since a sequential machine is deterministic its behavior -- the set of ordered pairs (w, w') (cf. end of section 1) -- is a function. Which functions, numerical or otherwise, can an f.t. represent? To answer this, we imagine an f.s.a. with <u>two</u> inputs which reads <u>pairs</u> of words symbol by symbol. Such an f.s.a. might be **defined** by rules

$$P\ q\ \binom{s}{s'}\ R \rightarrow P\ q\ R$$

where the pair $\binom{s}{s'}$ is an element of an <u>alphabet</u> of pairs. f.s.a.'s of this kind accept <u>pairs</u> of words. Any set of pairs definable in this way which is also a function***** which satisfies the property that the length of the input w is equal to that of w', and which is such that if $w = xy$ and $w' = uz$ and the length of x is that of u, then (x, u) is in the set as well, is representable by some f.t. We say that a function which is <u>length-preserving</u> and prefix <u>closed</u> is so representable. The converse also holds, yielding

<u>Proposition 9. A function is representable by (is the behavior of) an f.t. if and only if it is</u> f.s.a. <u>definable, length-preserving, and prefix closed</u>. (Elgot[4], and Elgot and Mezei[5]).

A little thought given to the conditions of representability shows that prefix closure and length preservation are almost obvious properties of functions representable by a strictly real time device, which is what an f.t. is.

Using these abstract results one can go on to specify the numerical functions within the powers of an f.t. By a more or less straightforward construction one can show that addition is representable, although since sums are in general longer than either addend, and the addend and augend are not necessarily the same length, certain adjustments in the alphabet must be made to satisfy the length preservation condition. Ritche[53] has shown.

<u>Proposition 10. The class of numerical function computable on a finite transducer includes the identity, successor and addition functions*</u> (above) <u>and is closed under composition. There is no</u> f.t. <u>multiplier</u>.

Propositions 9 and 10 pretty well summarize, in a general way, what can be done with a finite transducer.

Now that we have examined the behavioral capabilities of finite automata and Turing automata it is natural to ask whether there are any well understood models in between. By Church's thesis the Turing machine is the most powerful of all automata and the finite automata the weakest since anything weaker (one auxiliary symbol or one state) is merely a coding device. There are two answers, one for acceptors and generators and the other for transducers.

First, the theory of acceptors and generators has been quite richly developed but almost entirely from the point of view of mathematical linguistics -- which of course does not imply lack of value for the theory of automata. Two classes of generative grammar, in addition to the Turing generator and the finite state generator have been studied extensively, context sensitive grammars and context free grammars. The context sensitive ones are generators in our sense with rules $PgR \rightarrow Pg'R$ such that g is of the form $\phi_1 q \phi_2$ and g' is $\phi_1 x \phi_2$ for some words ϕ_1, ϕ_2 of $(SUQ)^*$ and for $x \neq \Lambda$. Thus the rules permit rewriting of auxiliaries in the context $\phi_1 - \phi_2$. The languages generated by such grammars are context sensitive. Although it is known that the theorems of the propositional calculus as well as the closed formulas of the predicate calculus (Hodes[27]) and also machine languages of digital computers are context sensitive, they are not of much interest to linguistics. Since to comment on this situation is beyond my scope I shall merely say that the phrase structure of sentences of context sensitive languages does not seem to be an appropriate model for understanding the structure of interesting natural languages like English or artificial ones like ALGOL or FORTRAN. They are of some slight interest in automata studies as seen below.

The context free grammars are like the context sensitive ones except that that $\phi_1 = \phi_2 = \Lambda$. An example is the set of rules for the ALGOL fragment we examined earlier, and, of course, the so-called Naur-Backus form of ALGOL is a context free language (Backus[1]). Much of English is context-free; e.g., many declarative sentences in the active voice and indicative mood. The context free languages may also be obtained by a set of theoretic definition in a way reminiscent of the regular sets (Ginsburg and Rice[20]).

A pushdown acceptor is an automaton like an f.s.a., but is equipped in addition with a storage tape which can be written on as well as read and which moves right and left with respect to the scanning head. It is arranged in such wise that only the last symbol written in the store is in a position to be read -- namely the store is a stack. Proposition 11 has been shown by Chomsky[12]; see also Schutzenberger[55].

Proposition 11. The language accepted by a pushdown acceptor is context free. The converse, however, is not true: some context-free languages require the more powerful resources of a non-deterministic pushdown acceptor. Such a device, in analogy to f.s.a.'s has a polygenic set of rules.

Proposition 12. A context-free language is accepted by some non-deterministic pushdown acceptor.

Unlike the f.s.a., equivalence between the deterministic and non-deterministic automata fails. Moreover as shown by Bar-hillel, Shamir, and Perles[2], the family of context-free languages is not closed under intersection. Thus the family is not Boolean. However, the intersection of a context-free language with a regular set is again context free.

A linear bounded acceptor (l.b.a.) is a Turing acceptor with bounded tape; the

length of the tape (which is either fixed or which can grow up to the bound) is a linear function of the input length. A pushdown acceptor is really a species of l.b.a. since, as a little thought will show, the latter if supplied with a tape (roughly) twice the length of a given input can use the extra stretch as a stack. Moreover the l.b.a. accepts context sensitive languages (Landweber[34]).

Proposition 13. The languages accepted by a linear bounded acceptor are context sensitive.

The converse, to my knowledge, is an open problem. Moreover it is an open problem whether the non-deterministic l.b.a., defined by relaxing the usual monogenicity requirements on Turing rules, is associated to the same languages as the ordinary l.b.a. Landweber has also obtained the surprising conclusion -- surprising in view of the contrary results for context-free languages -- that the context sensitive languages are closed under intersection. It is an open question, at this writing, whether the family is Boolean.

Chomsky[11] has shown the recursiveness of context sensitive languages. The same holds, by inspection of the rules, for the context free. Finally, the families of regular sets, context free languages, context sensitive languages, and recursively enumerable sets in that order form a simple hierarchy with respect to proper set inclusion. Obviously the associated automata form a hierarchy as well.

The sketch of the last ten paragraphs hardly does justice to the amount of work reported in the area of languages and automata, of linguistics and automata theory. For a thorough-going survey with bibliography as of 1963 see Chomsky[10].

We now return to the question posed earlier about transducers. Almost nothing is known about these automata in the region between sequential machines and Turing machines. I will mention a few isolated facts and then return to transducers.

Schutzenberger[56] has examined compositions of sequential machines such that the output of one machine is the input to another. Also in his machines an input symbol s as in rule (2) above may be replaced by a word on the output alphabet 0^{+}. He shows that the behavior of such machines (i.e., the set of pairs w, w'), where w is an input and w' the output of the composed machine) is closed under finite composition, and that these machines transform regular sets into regular sets. By specialization, therefore, the same holds for finite transducers (see also Ginsburg and Rose[21].

Oettinger[50] has shown that the pushdown transducer, which is like an acceptor except that the input tape may be written on as it passes through the machine -- precisely as in a finite transducer --, is capable of translating parenthetic context-free languages into parenthesis-free ones and conversely.

Ritchie[53] has characterized the numerical functions computable on a linear bounded transducer the latter being precisely a Turing machine except for tape bounded as in the case of the l.b.a. Roughly speaking this result says that the class of functions computable by these machines includes addition, the identity function, the successor function, and multiplication (See (1)) and is closed under composition, transformation of n-variable functions to m-variable functions, $n \geq m$, by replacement of variables by constants, and primitive recursion. The last operation is, however, permitted only when the next earlier function value f(y) used with y in defining f(y + 1) satisfies certain boundedness conditions.

It can be shown that sorting, that is the function f on S^* into $s_o^* s_1^* \ldots s_n^*$, where S is a simply ordered set, such that f takes each word into "sorted order", requires at least a linear bounded transducer. (Nelson[49]).

Structure

It would be highly misleading to suggest that any meaningful exposition of the recent work on automata structures could be attained without using a considerable amount of elementary but abstract and technical mathematics, chiefly algebra. Presupposing a minimal knowledge (as we already have in using set theory and Boolean algebra) we may, however, be able to sketch some of the recent main concepts. All of the results we mention are about finite state acceptors and transducers. We omit explicit discussion of state minimization, state identification experiments and realization of automata by switching nets since these have been already discussed extensively in books (Gill[17], Ginsburg[18]) and many papers, especially Burks and Wang[7]. Interesting as the problems are, we must omit here discussion of structural problems of switching and iterative nets; Burks and Wright[8] and Holland[28] are representative. What remains for discussion will not be exhaustive either.

The defining rules (7) (modified to specify exactly one rule $P\ q\ s\ R \longrightarrow P\ O\ q'\ R$ for each pair $(q, s) \in Q \times S$) of sequential machine together with provisos suggest that we regard these machines as embodying two functions M: $Q \times S \longrightarrow Q$ and the other N: $Q \times S \longrightarrow O$. Thus we may take a sequential machine to be an ordered sextuple $< S, O, Q, q_o, M, N >$ where S and O are alphabets and Q is the set of states (auxiliaries) belonging to a given machine; $M(q, s) = q'$ if and only if $P\ q\ s\ R \longrightarrow P\ O\ q'\ R$ is a rule of the given machine for some $o \in O$; and $N(q, s) = O$ if and only if $P\ q\ s\ R \longrightarrow P\ O\ q'\ R$ is a rule of the given machine for some $q' \in Q$. q_o is the initial state, M is the transition function, and N is the output function.

By a similar technique we may set up an acceptor as a mathematical system. Since in this case there is no output, there is no N function. Instead we have a subset of final states Q'. So we can consider an acceptor to be a quintuple $< S, Q, q_o, M >$ characterized mutatis mutandis as above.

Our interest here will be in structural properties of either sequential machines or acceptors with respect to the M function alone, and for the time being we will not even concern ourselves with the accepting behavior of an f.s.a.; so we will ignore final states. This leads us to abstract from transducers and acceptors and to consider transition systems. A transition system is a quadruple $< S, Q, q_o, M >$ where S and Q are any non-empty, finite, disjoint sets, and where M is any arbitrary function M: $Q \times S \longrightarrow Q$.

As before S^* is the set of words on S. It is easy to see that since S^* contains all finite words, it is algebraically closed under the concatenation operation, which we will temporarily denote by "o". Also, o is an associative operation, and since $x\ o\ \Lambda = \Lambda\ o\ x = x$ for any x, we have an identity element. Hence, the triple $\delta = < S^*, o, \Lambda >$ is a semi-group with identity (alternatively a monoid). Moreover, it is free, which means roughly that no equalities other than $x = x$ (or $x = \Lambda\ x = x\ \Lambda$) hold in the system. The function M is extended to elements of S^* as follows

$$M(q, \Lambda) = q$$

$$M(q, xs) = M(M(q, x), s)$$

Formally, $M(q_o, x) = q$ if and only if q is a theorem of an axiom $q_o x$ in, say, an acceptor. Intuitively it is the state a transition system goes into from the initial state. We may assume without losing anything that for our systems every state q is a terminal result for some input: $M(q_o, x) = q$.

Next, we are interested in equating those words which always take the transition system from the same states to the same states. Thus we introduce an equivalence relation E, as follows:

$$x \, E y \quad \text{if and only if for every } q, \; M(q, x) = M(q, y).$$

Thus x is E-equivalent to y, if and only if the indicated equation holds for all q. It is obvious that E is properly termed an equivalence relation since $x \, E \, X$; $x \, E \, Y$ implies $y \, E \, x$; and $x \, E \, y$ and $y \, E \, z$ implies $x \, E \, z$. E determines a partition of S^* into equivalence clauses. Moreover E satisfies the congruence or substitution property with respect to concatenation.

$$\text{If } x \, E \, y \text{ and } z \, E \, w, \text{ then } x \, z \, E \, y \, w.$$

This is easily proven from the definition of M and of E, since

$$M(q, xz) = M(M(q, x), z) = M(M(q, y), z) = M(M(q, y), w) = M(q, yw)$$

In algebraic studies a property of congruence classes (equivalence classes determined by a congruence relation) is that they define quotient systems. In the case of our free semi-group δ, if T is a transition system for which the congruence relation E has been defined, then

$$\delta/T = \langle [E] \, , \, o, \, [\Lambda] \rangle \, ,$$

where $[E]_T$ is the set of equivalence classes $[X]_T$ determined by E, is a quotient semi-group with identity. Here $[x]_T$ is the set of all words $y \in S^*$ such that $x \, E \, y$ where the M function of course belongs to T. δ/T is a semi-group since $[x] \, o \, [y] = [x \, y]$ and o is associative. $[\Lambda]$ is its identity element.

δ/T is also called the semi-group of the transition system T. Proposition 14. To every transition system T (hence, every automaton with an M function, including a Turing machine) there corresponds a semi-group δ/T.

Another well-known fact from algebra is that a quotient system is a homomorphic image of its parent system. Thus in the case of semi-groups $\langle A, \cdot, e \rangle$ and $\langle A', o, e' \rangle$ a function $f: A \rightarrow A'$ is a homomorphism provided that

$$f(e) = e',$$

and

$$f(a \cdot b) = f(a) \; o \; f(b), \text{ for all } a, b, \in, A.$$

Specifically for δ and δ/T, the function $\phi : S^* \rightarrow [E]_T$ determined by $\phi(x) = [x]_T$ is a homomorphism since

$$\phi\ (\Lambda) = [\Lambda]_T \text{ by definition,}$$

and

$$\phi\ (x\ y) = [x\ y]_T = [x]_T \circ [y]_T = \phi\ (x) \circ \phi\ (y).$$

A semi-group isomorphism is a homomorphism f such that f is a one to one correspondence (the sets of the semi-groups are equinumerous) and the image of the function f is all of the semi-group elements: i.e., the function is onto.

A very interesting fact, discovered by Myhill[43], is the following.

Proposition 15. Every finite semi-group with identity is isomorphic to the semi-group $\mathcal{S}/T$ of a transition system T.

This theorem shows the close connection between the whole of semi-group (including group) theory and automata.

So much for the bare essentials. The beauty of the treatment is that algebraic methods can now be applied to the structural problems of automata. It can be shown, for example, that if $\mathcal{S}/T$ is a group then T is a permutation system, namely that a state q in $M(q, x) = q'$ is unique for every q' and every x. In plainer English, you can always tell what state q the transition system in state q' came from, given an arbitrary input (Burks and Wang[7]). Many other algebraic ideas are used to discuss automata on the basis of the automaton semi-group (e.g., Beatty[4], Mezei[39]).

Another connection with algebra is established by viewing the system of all automata (again using the abstraction to transition systems) as a lattice.

A lattice is essentially a generalization of a Boolean algebra. In a lattice it does not necessarily hold that every element has a unique complement, nor do the usual associative principles necessarily hold $\dot{+}$ $\dot{-}$. To show that the family of transition systems is a lattice we aim at showing that the family is isomorphic to another system already known to be a lattice.

Similar to E above, we have now an equivalence relation R defined on S^*, called a right invariance relation:

$$x\ R\ y \text{ if and only if } M(q_o, x) = M(q_o, y),$$

which says that x and y are R-equivalent if they lead to the same state from the initial state q_o. The property right invariance says that if x R y then x z R y z --

if x and y lead to the same target state, then so do x and y with a common tail attached. This property is easily proved from the definition of R.

Now consider all of the distinct relations R_i determined by transition systems. It can be shown that this family, which we will call P_i, forms a lattice where the interpretation is as follows: $R_i \subseteq R_j$ means that every pair constituting the relation R_i is a pair of R_j; $R_i \cap R_j$ (the lattice intersection operation or meet) is the set of pairs in both R_i and R_j; and $R_i \cup R_j$ (the lattice union operation or join) is the set of pairs built up by starting with the pairs in the ordinary set union of R_i and R_j and then adding (x, z) to the set if (x, y) and (y, z) are already in it, and so on.

Next, we want to build up a family T of transition systems corresponding to P. Two transition systems $T = < S, Q, q_0, M >$ and $T' = < S, Q', q'_0, M' >$ (note common alphabet, S) are homomorphic if there exists a function $\phi : Q \to Q'$ such that

$$\phi\ (q_0) = q'_0$$

and

$$\phi\ (M(q, x)) = M'(\phi\ (q)\ x),\ x \in S^*,\ q \in Q.$$

When ϕ is one-one onto Q', then T and T' are isomorphic and ϕ is an isomorphism, otherwise a homomorphism.

Clearly two isomorphic systems will determine one invariance relation R, since if $M(q_0, x) = M(q_0, y)$ holds, then surely $\phi\ M(q_0, x) = \phi\ (M(q_0, y))$; and since ϕ is one-one the converse holds. Similarly for any number of isomorphic systems. So there is a one-one correspondence between relations R_i and an isomorphism type, T_i; i.e., a set of mutually isomorphic transition systems. The family T has the types T_i as elements.

Now P is a partially ordered system with respect to $\subseteq$ (since it is a lattice). Also T is partially ordered with respect to the relation homomorphism. Further, two partially ordered systems O and O' with relations α and β are order isomorphic if there is a one-one correspondence θ between O and O' such that

$$a\ \alpha\ b \quad \text{if and only if } \theta\ (a)\ \beta\ \theta \qquad (b)$$

for any a, b, $\in$ o. It can be shown that P and T are order isomorphic with respect to the relations $\subseteq$ and homomorphism. Finally if P is a lattice and is order isomorphic to T, then T is a lattice.

Proposition 16. The family T of transition systems is a lattice isomorphic to the lattice P of right invariant relations.

On the basis of this result it is possible to discuss (among other things) in an exact way the problem of decomposing an automaton into sub-automata which operate in parallel, in cascade, or in cascade, or in certain feed-back arrangements. The essential idea in these applications is to find decompositions via equivalence relations, on the set of states of a transition system, having the congruence property. One has to select equivalence relations on Q, $\equiv$ let us say, so that

$$q \equiv q' \text{ if for all } x,\ M(q, x) \equiv M(q', x).$$

A familiar example is the output equivalence relation used in finding minimal-state transducers. Parallel or cascade systems exist when two or more quotient systems, determined by the appropriate congruence relation, operating synchronously and beginning in initial states, are isomorphic to the union of these transition systems in the lattice of transition systems. Numerous developments of these ideas have been developed by Hartmans[23] The main outlines of the above development are from Büchi[5].

Another approach to the understanding of sequential machines has been made by Krohn and Rhodes[33] who in effect study the machine as embodying the function N: $Q \times S \to Q$ extended to sequences of S^* so that $N(q, \Lambda) = \Lambda$ and $N(q,x\ s) = N(m(q, x),s)$. Their approach is deep and general and finds structural properties into which one may inquire from the point of view of (non-elementary) group theory.

Solvability Results and Perspectives

The above sections sketch what I believe to be the main lines of growth of the theory about behavioral and structural problems.

Another aim is to understand which problems about automata can be solved by automata. We say that a problem is recursively solvable if that problem converted into one about the recursiveness of sets is answered by a proof that the set in question is recursive. In machine terms this can be interpreted in any number of convenient ways: we may show Turing computability of the associated function of the set; or we may show the set to be definable by an l.b.s. (or less), etc.

In general the questions about Turing machine, it occurs to one to ask, are unsolvable. Most of the required proofs depend on the fact that there exist recursively enumerable sets whose complements are not recursively enumerable. So the sets in question are not recursive. Hence there are sets accepted by or generated by Turing machines which are not recursive. It follows more or less directly from this circumstance that the problem whether a given Turing machine with given initial word, will ever come to a terminal result, or halt, is unsolvable. Similarly it is unsolvable whether two Turing machines are equivalent in the sense that they accept (or generate) the same sets or compute the same functions. The same holds of programs; and moreover, the problem whether a program computes what it is supposed to is unsolvable -- in particular no program can decide whether a given program will do what is desired of it.

At the other end of the simple hierarchy of machines, the problems tend to be recursively solvable and sometimes trivially so. Not only are the regular sets recursive, but there is an algorithm for designing an acceptor, given a regular set and conversely there is an algorithm for finding the defined regular set, given an f.s.a. (Kleene[32], McNaughton, Yamada[38]). There is an algorithm for equivalence of f.s.a.'s and also for f.t.'s, and also for finding reduced transducers (Moore[42]). There are algorithms for finding decompositions of transducers, for detecting whether a transducer is connected (every state entered from the initial state by some input), or strongly connected (every state into every other state by some input). There is, for such elementary machines, with initial states, a recursive procedure for telling whether for any two automata of the same type and any function f, whether f is a homomorphism. The question whether there are any words acceptable by an alleged f.s.a. is recurisvely solvable (emptiness problem).

For the automata of intermediate powers the known results tend to be negative. A large number of such problems is surveyed in Chomsky[10] based on the work of Bar-Hille, Perles, and Shamir[2]. However, the emptiness problem for context-free grammars

is solvable, as is the problem whether the language generated by such a grammar is infinite. The unsolvable problems include the following for both the case of the intersection of two context-free languages or the complement of such languages; whether the intersection or complement is empty, finite, another context-free language, or a regular set. Finally the ambiguity problem, namely whether there is more than one derivation of a given context-free language -- such as ALGOL --, is unsolvable (Cantor[9]). Attempts have been made, owing to the unsolvability of this problem, to redesign ALGOL as an unambiguous language (Johnson[30]).

In the case of transducers the results are few and far between. Ginsburg and Hibbard[19] have shown that if R_1 and R_2 are regular sets, the problem whether there is a finite transducer which maps R_1 and R_2 is recursively solvable; on the other hand, the companion problem for context-free languages is unsolvable. There is no algorithm for deciding whether there exists a finite transducer such that it (Ginsburg and Rose[22]) maps one context-free language into another. When there is such a transducer, however, the output, given a context-free language put in sentence by sentence, will again be context-free (Ginsburg and Rose[21]). There are no such analogous results for transducers of more power than an f.t.

As mentioned earlier the most rapid development of the theory has been in the section related to linguistics -- the theory of generators and acceptors. I would like, finally, to discuss what appear to me to be the chief problems for future inquiry. Almost all of these problems center around more complete understanding of transducers; i.e., of computations in the broadest sense of the word. Briefly, what transformations (calculations, cognitions, translations, derivations, inference, game-plays) are possible and with how much stuff? There are, it seems to me, five overlapping problem areas worth discussing.

(1) - To what extent are automata models in our sense (formal rules) adequate for studying natural phenomena which we intuitively associate with "control" rather than "energy transformation"? The following list suggests possible areas of application: self reproduction (von Neumann[60], Burks[6] and Myhill[44]); non-stochastic learning (Miller and Chomsky[40]); genetic coding; language processing, human behavior, and thought; control in biological and social organisms. My view is that the study of such phenomenon from the point of view of formal rules is relevant and may be fruitful, while a study emulative of hard physical science, usually by way of statistics, is likely to remain barren. However, see (5) below.

(2) - None of the transducers discussed here are adequate models of digital computers. If an input tape is part of a computer, then a general purpose computer is not even a finite transducer, since an existing computer could not possibly add arbitrarily large integers, while, as we have seen, finite transducers can. It is safer (and less "trivial-sounding") to assume prior to discussion of any formal model that a computer has "in principle" all the memory it needs. If so, it is a Turing machine. A still better makeshift is to assume that a computer does not have indefinitely expansible memory, but always just enough for in-putting a problem (linear function of the input). Thus, the most adequate model would be a linear bounded transducer. However, there is no model of a "universal" l.b. transducer; yet we think of stored program computers as "universal."

When we examine structure it is less clear what a real computer or computer circuit is. Various models have been proposed by Elgot and Robinson[16], Holland[29] and in a related way Hennie[25], all of which are closer to the real thing than the abstract sets of rules we have discussed. All of them, however, are behaviorally equivalent to a Turing machine or less.

(3) - It is known as we have seen, which functions are Turing computable and also which ones are finite transducer computable. Ritchie[53] and many others have considered more or less reasonable hierarchies of (primitive) recursive functions corresponding to which one "in principle" can associate, crudely, transducers of various powers. Hartmanis and Stearns[24] have discussed complexity of computations on Turing machines with respect to such parameters as the time it takes to compute. All of these approaches to the problem of ordering computations according to complexity in some sense seem doomed to eternal commerce with Turing machines. It seems to me that a fruitful, but certainly difficult, point of view is the following. Starting with finite sets, any function from one to the other can be realized by a finite transducer. Now let us consider infinite regular sets. There is a non-denumerable set of functions from one to the other, and of these the length-preserving, **prefix-**closed ones can be realized by finite transducers. Which ones can be realized by machines of greater powers? Similarly for transformation of context-free languages to regular sets, or the reverse, and so on.

It seems to me that such studies might provide insight into non-numeric computation; i.e., symbol manipulation or information processing. There is no <u>theory</u> of such computation at all except in the sense of associating the non-numeric to the numeric via Gödel numbering (Davis[13]). But such association does not tell us, for example, a thing about <u>non-numeric computational equivalence</u>. There are non-erasing machines (Wang[61]) which can compute any partial recursive function; but they obviously cannot perform any computation calling for clean, readable tape. If w is inscribed on tape with blank elsewhere, for example, no such machine can produce w^1 with blank elsewhere, where w^1 is w written in reverse. It follows that a theory of numerical computation is not adequate for the non-numeric -- which is obvious but needs to be said anyhow.

I claim nothing is known about most algebraic translations one would like to make automatically in the sense that we know; e.g., what kind of transducer short of a Turing machine it takes to transform a polynomial into nested form for economical computation, or "analytically" to integrate elementary functions. In this area programmers have designed special methods for symbolic calculation (e.g., McCarthy[36]) but this is not the same as a theory in the sense of the study of automata, (which may merely hint at a severe limitation of that subject, as it now stands).

(4) - A socially and philosophically important problem concerns the relative powers of human beings and computers for selected cognitive tasks. This is obviously no joke, since not only many clerks but also computer coders are either out of jobs, or have had to learn systems programming, or have become administrators. It has been suggested that Gödel's theorem (which can not be described in any way but a misleading way in a short space) shows there are truths human beings can discover which are not open to computers (Nagel and Neuman[46], Lucas[35]). In my opinion Gödel's theorem shows no such thing, and the field is wide open.

In strict terms, these task problems call for adequate automaton models of minds and machines and an understanding of the functions realizable by each model. A further question, going beyond the "selected tasks," is whether non-formal, non-deductive activities we attribute to human beings are achievable by <u>computers</u>. Such a question seems to transcend automata theory as such.

(5) - A problem perhaps of more concern to the logician than **to the computer** scientist concerns the adequacy of Church's thesis +++ ---. If either there are recursive functions which are intuitively not effectively calculable or there are effectively calculable functions which are not recursive, then either the thesis or its converse is false.

In the computer field one would like to model actual computers, as in problem (2) as closely as possible. If an actual computer is only a l.b. transducer then one might argue that the converse of the thesis is false. From this practical point of view, many problems of interest to computer science heretofore believed effectively solvable (and recursively solvable) are in fact not. It is entirely possible that one could get into serious trouble trying to program a Turing computable algorithm on a mere computer. Certainly, in every case a sign of trouble would be exhaustion of storage, but there is no way of predicting such an outcome in advance. Clearly this kind of problem is intimately connected with the hierarchy and complexity questions mentioned in (3).

On the other hand, there seems to be some evidence that the thesis is false (Heyting[26]). From our viewpoint here any such evidence would also be evidence that our sufficiency condition for automata (considered now as a full definition) is too restrictive. If we include probabilistic machines, machines which modify their own defining rules, and the like, we may still retain the intuitive notion of a mechanical procedure while increasing powers of computation beyond the recursive functions.

A related problem concerns computer models for computability of real functions (or approximation thereto) or for analog computation. Almost nothing has been done along these lines; however, the interested reader might want to check Myhill[45] or Montague[41]. The work of Zadeh and DeSoer[63], appears to suggest parallels between systemic structures of continuous control (i.e., analog) devices and automata; however, the same can not be said about behaviors.

---

[1] Backus, J. W., "The Syntax and Semantics of the Proposed International Algebraic Language of the Zürich ACM-GAMM Conference," *ICIP Paris;* June, 1959.

[2] Bar-Hillel, Y., Perles, M., and Shamir, E., "On Formal Properties of Simple Phrase Structure Grammars," Technical Report No. 4, *USONR Information Res. Branch,* Jerusalem; July, 1960.

[3] Bar-Hillel, Y., and Shamir, E.," Finite State Languages: Formal Representation and Adequacy Problems, *Bull. Res. Council of Israel,* p. 155-166; 8F (1960).

[4] Beatty, J. C., "On Some Properties of the Semi-Group of a Machine Which are Preserved under State Minimization," *IBM Report;* May, 1964.

[5] Büchi, J., "Mathematical Theory of Automata," Unpublished notes of lectures by Büchi and J. R. Wright, *Univ. of Michigan;* 1960.

[6] Burks, A., "Cellular Automata," Engineering Summer Conferences, *Univ. of Michigan;* 1963.

[7] Burks, A. and Wang, H., "The Logic of Automata," *J. ACM,* p. 193-218, 279-297; 4/1957.

[8] Burks, A. and Wright, J., "Theory of Logical Nets," *Proc. IRE,* p. 1357-1365; 41/1953.

[9] Cantor, D. G., "On the Ambiguity Problem of Backus Systems," *J. ACM,* p. 477-479; 9/1962.

[10] Chomsky, N., "Formal Properties of Grammars," *Handbook of Mathematical Psychology,* Wiley, Vol. 2; 1963.

[11] Chomsky, N., "On Formal Properties of Certain Languages," *Inf. and Control,* 137-167; 2/1959.

[12] Chomsky, N., "Context-Free Grammars and Pushdown Storage," RLE *Quart. Prog. Report.* MIT; March, 1962.

[13] Davis, M., "Computability and Unsolvability," *McGraw-Hill;* 1958.

[14] Elgot, C. C., "Decision Problems of Finite Automata and Related Arithmetics," *Trans. Amer. Math. Soc.,* p. 21-51; 98/1961.

[15] Elgot, C. C., and Mezei, J. E., "On Relations Defined by Generalized Finite Automata," *IBM J. Res. and Dev.,* 47-68 9/1965.

[16] Elgot, C. C. and Robinson, A., "Random-Access Stored-Program Machines, an Approach to Programming Languages," *J. ACM,* p. 365-399; 11/1964.

[17] Gill, A., "Finite State Machines," *McGraw-Hill;* 1962.

[18] Ginsburg, S., "An Introduction to Mathematical Machines Theory," *Addison Wesley;* 1962.

[19] Ginsburg, S. and Hibbard, T. N., "Solvability of Machine Mappings of Regular Sets to Regular Sets," *J. ACM,* p. 302-312; 11/1964.

[20] Ginsburg, S. and Rice, H. G., "Two Families of Languages Related to ALGOL," *J. ACM,* p. 350-371; 9/1962.

[21] Ginsburg, S. and Rose, G. F., "Operations Which Preserve Definability in Languages," *J. ACM,* p. 175-195; 10/1963.

[22] Ginsburg, S. and Rose, G. F., "Some Recursively Unsolvable Problems in ALGOL-Like Languages," *J. ACM,* p. 29-47; 10/1963.

[23] Hartmanis, J., "Symbolic Analysis of a Decomposition of Information Processing Machines," *Inf. and Control,* p. 154-178; 3/1960.

[24] Hartmanis, J. and Stearns, R. E., "On the Computational Complexity of Algorithms," *Trans. Amer. Math. Soc.,* p. 285-306; 117/1965.

[25] Hennie, F. C., "Iterative Arrays of Logical Circuits," *MIT* Press; 1961.

[26] Heyting, A. (Ed.), "Constructivity in Mathematics," North-Holland Pub. Co.; 1959.

[27] Hodes, L., "Context-Free Languages and Logic," *IBM Res. Report;* 1964.

[28] Holland, J. H., "Iterative Circuit Computers," *Proc. West. Joint. Comp. Conf.,* p. 259-265; 1960.

[29] Holland, J. H., "A Universal Computer Capable of Executing an Arbitrary Number of Sub-programs Simultaneously," *Proc. Eastern Joint Comp. Conf.,* p. 108-113; 1959.

[30] Johnson, J. B., "A Class of Unambiguous Computer Languages," *Comm. ACM,* p. 147-148; 8/1965.

[31] Keller, H. B., "Finite Automata, Pattern Recognition and Perceptions," *J. ACM,* p. 1-20; 8/1961.

[32] Kleene, S. C., "Representation of Events in Nerve Nets and Finite Automata," *Automata Studies,* Princeton Univ. Press; 1956.

[33] Krohn, K. B. and Rhodes, J. L., "Algebraic Theory of Machines," *Mathematical Theory of Automata,* Polytechnic Press; 1963.

[34] Landweber, P. S., "Three Theorems on Phrase Structure Grammars of Type I, *Inf. and Control,* p. 131-136; 8/1963.

[35] Lucas, J. R., "Minds, Machines, and Gödel," in *Minds and Machines.* Prentice-Hall; 1964.

[36] McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine," *Comm. ACM,* p. 184-195; 3/1960.

[37] McNaughton, R., "The Theory of Automata; a Survey," *Advances in Computers.* Academic Press; 1961.

[38] McNaughton, R. and Yamada, H., "Regular Expressions and State Graphs for Automata," IRE *Trans. on Elec. Comp.,* p. 39-57; 9/1960.

[39] Mezei, J., "Structure of Monoids with Application to Automata," *Mathematical Theory of Automata,* Polytechnic Press; 1963.

[40] Miller, G. A. and Chomsky, N., "Finitary Models of Language Users," *Handbook of Mathematical Psychology,* Vol. II. Wiley, 1963.

[41] Montague, R., "Towards a General Theory of Computability," in *Logic and Language.* D. Reidel Pub. Co., Dordrecht, Holland; 1962.

[42] Moore, E. F., "Gedanhen Experiments on Sequential Machines," *Automata Studies,* Princeton Univ. Press; 1956.

[43] Myhill, J., "Finite Automata, Semi-Groups, and Simulation," *Engineering Summer Conferences, Univ. of Michigan;* 1963.

[44] Myhill, J., "Self-Reproducing Automata," Engineering Summer Conferences, *Univ. of Michigan;* 1963.

[45] Myhill, J., Nerode, A. and Tennenbaum, S., "Fundamental Concepts in the Theory of Systems," *WADC Report;* 1957.

[46] Nagel, E. and Newman, J. R., "Gödel's Proof" New York Univ. Press; 1958.

[47] Naur, P. (Ed.), "Report on the Algorithmic Language ALGOL 60," *Comm. ACM*, p. 299-314; 3/1960.

[48] Nelson, R. J., "Introduction to Automata," *Lecture Notes/ Case Institute of Technology;* 1965.

[49] Nelson, R. J., "Ordering Automata;" forthcoming in *Comm. ACM.*

[50] Oettinger, A., "Automatic Syntactic Analysis and the Pushdown Store," *Proc. Symp. in Appl. Math. of Amer. Math. Soc.*, p. 104-129; 12/1961

[51] Rabin, M. O., "Probabilistic Automata," *Inf. and Control*, p. 230-245; 6/1963.

[52] Rabin, M. O. and Scott, D., "Finite Automata and Their Decision Problems," *IBM J. Res. and Dev.*, 3/1959/: p. 114-125; 3/1959.

[53] Ritchie, R. W., "Classes of Predictability Computable Functions," *Trans. Amer. Math. Soc.*, p. 139-173; 106/1963.

[54] Rogers, H., Jr., "The Present Theory of Turing Computability," *J. SIAM*, p. 114-130; 7/1959.

[55] Schutzenberg, M. P., "On Context-Free Languages and Push-Down Automats," *IBM Research Report;* 1962.

[56] Schutzenberger, M. P., "A Remark on Finite Transducers," *Inf. and Control*, p. 185-196; 4/1961.

[57] Shannon, C. E., "A Universal Turing Machine with Two Internal States," *Automata Studies*, Princeton Univ. Press; 1956.

[58] Shepherdson, J. C., and Sturgis, H. E., "Computability of Recursive Functions", *J. ACM*, p. 217-255; 10/1963.

[59] Turing, A. M., "On Computable Numbers with an Application to the Entscheidungs Problem," *Proc. London Math. Soc.*, p. 230-265; 42/1936: p. 544-546; 43/1937.

[60] Von Neumann, J., "Lecture Notes on Reproducing Machines"; To appear in a book, A. Burks, Ed.

[61] Wang, H., "A Variant to Turing's Theory of Computing Machines," *J. ACM*, p. 63-92; 4/1957.

[62] Watanabe, S., "5-Symbol 8-State and 5-Symbol 6-State Universal Turing Machines," *J. ACM*, p. 476-483; 8/1961.

[63] Zadeh, L. A. and Desoer, C. A., "Linear System Theory, The State Space Approach," *McGraw-Hill;* 1963.

---

* So named after A. M. Turing[59]. For a description of Turing Machines see Davis[13].

** This is known as Church's Thesis; see Davis[13].

*** Also customarily called a finite automaton. We have departed from this usage to preserve "automaton" as a generic term.

**** Q x S is the set of all ordered pairs (q, s) such that $q \in Q$ and $s \in S$.

***** It is customary to define a function on a set A with range B as the set of ordered pairs $(a, b) \in A \times B$ such that no two distinct pairs have the same left component. This insures single-valuedness.

± He refers to the mapping represented by the composed machines as a "transduction," which is of course a more specialized usage of the word than ours.

±± Here again we must assume knowledge of the elementary facts.

±±± Church's thesis (section 1 above) says that all effectively calculable (i.e., algorithmic) functions of non-negative integers are recursive.