Only in the former case is there a direct correspondence between the information and the entropy; namely (using Δ to stand for a change in a quantity)

$$\text{Bound information} = -\Delta \text{ entropy} = +\Delta \text{ negentropy}$$

The main point of impact on the present argument is in connection with the act of retrieving or inscribing 'information' as, for example, retrieving the 'information' in a description such as a table. Such a thing does not *have* information; it has a pattern. This pattern may convey information to you or I, or any appropriate computing engine that can interpret the language in which it is written. In this sense the described pattern has *potential information to a user* which is obtainable (a) if the user can interpret what is inscribed and can (b) provide the negentropy needed to read the description and thus engage in a transaction. Similar comments apply to making an inscription in the first place. For this reason pains were taken to calculate information indices over occurrences or events. There is only information in a contingency table, recording events, in so far as there is a user.

The argument is not essentially concerned with physics and mechanics as such. Some parts of it are, however, very much concrened with the processes involved in using information and the coherency, or at any rate the synchronicity, with which information in a description is activated and used to determine operations or to reduce uncertainty on the part of sentient beings. These remarks are thus properly made at the outset, though they mainly bear upon the argument in the next volume.

2 Machines

Most engineers think of a machine as a tangible entity such as a lathe or a diesel engine. Naturally they include telephone exchanges and computers as machines also, so that 'machineness' does not depend in an essential way upon energetic transformations; to crunch numbers in data processing is as legitimate a machine activity as crunching rock from a quarry. All the same the engineers' image has a healthy tang of materiality about it and at a later stage we shall retrieve the materiality as a general notion of 'efficiency in embodiment and execution'. But at first it is more profitable to look at the other side of the coin and to emphasise the 'process' inherent in a machine and machine organisation.

1 Abstract Machines

In sharp contrast to the engineer, Ashby captures the essence of a machine in the abstract; a point of view that is compatible both with the engineer, and abstract automaton theory. All of the machines under discussion should, until further notice, be regarded as abstract automata but Ashby's treatment of the subject, which the serious student should read (Ashby, 1964b; reprinted in Stewart, 1967) is more general than most and has a built-in guarantee of realisability (find the guarantee, as it is a non-trivial problem).

1.1 The essence of the simplest type of abstract machine, a state-determined machine, is that it constitutes a 'coding of simple succession'.

The crux of Ashby's definition lies in the following concepts:

1. One and only one state occurs at a time. It being assumed that an observer or designer can point out, independently, the system which is said to have states, so that states are sets of (exclusive and exhaustive) alternatives, $z \in Z$, where Z is the *state set* (of whatever is observed to be a machine).
2. That the constitution of Z is unchanging over the interval of the observation.
3. A state is a complete account of the condition of whatever is observed; if it happens that certain properties were manifest in advance of the states,

then a state corresponds to their conjoint values and is expressed by a conjunctive expression (of a language in which these properties are named as unary but (in general) many valued predicates) that refers to all these predicates.

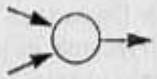
4. The notion of 'at a time' is crystallised by the idea that 'succession' is represented by an ordering index with a successor operation; for example, the index of the integers or the real numbers $\tau \in T$ . . .

5. It is postulated (a) that there is a machine clock, beating out machine time units τ (for example, as impulses from an oscillator driving a shift register); (b) that any observer has a stopwatch, indicating *his* time units which we refer to as $t = 0,1$ . . .; (c) there is a one to one correspondence relating τ and t (with due respect to units of measurement for the τ clock might be fast or slow or even variable) that allows for synchronisation between t and τ; and (d) as a waiver, which will be useful later, there may be several machine clocks but if so they are synchronously operated (for example, one clock may start when another clock stops or the pace of one clock may depend upon a master clock). Further, there may be several submachines with different clocks. But, if so, there exists a (cartesian) product machine with states that are a product of the states of the several submachines such that, whatever the clocking characteristics of the submachines, only one state of the machine occurs at once.

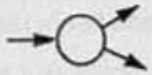
Under these circumstances a state determined machine in isolation is specified by a function or rule

$$Ru; Z \rightarrow Z \text{ or } Z(\text{next}) = Ru\,(Z \text{ existing})$$

Since *Ru* is a function, a picture such as



in which nodes represent states and the arcs stand for changes of state, is perfectly legitimate, whereas the following is not.



As a result, a machine occupying an initial state at some fiducial value '$\tau = 0$' of τ will have a state trajectory $\langle z_0, \ldots, z_\tau \rangle$ that leads either to an equilibrial state or to a cycle of states. The particular terminal condition may or may not depend on the initial state but usually it does so. Hence a complete specification of the machine is given by

$$Ru; Z, T \rightarrow Z$$

or given the assumptions which have been made by

$$Ru; Z \rightarrow Z \text{ and } z = z_0 \text{ at } \tau = 0$$

It will be noted that in any state-determined system the successor operation is redundant, which restates Ashby's cannon.

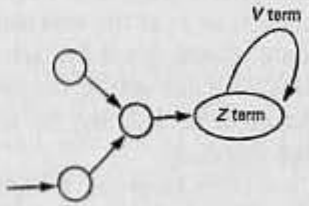
The crux of the matter becomes transparent if discrete *outputs* are assigned to the machine by associating certain of the state transitions with labels, usually numerical labels, v_τ (including a label *null* or no output), so that a state trajectory, $\langle z_0, \ldots, z_\tau \rangle$, might be rewritten as a behaviour (namely, a description of the state trajectory, obtained by labelling) such as $\langle v_0, \ldots, v_\tau \rangle$. Since *Ru* is a function and since it does not change, there can be no more labels than states. Usually there are fewer labels. Hence the mapping 'Output', namely

$$Output; Z \rightarrow V; \text{ (defined if } z \text{ changes otherwise } v = null).$$

is usually many to one.

Suppose that there are two labels 1 and Null; then periodically *V* assumes the value '1' and the machine is an oscillator. Suppose that a one-to-one correspondence is established (using numerical tags) then the machine is a clock with state numbers on its display.

1.2 As I have chosen to define it, a machine is a dynamic entity. The only sense in which it stops or halts is that its state trajectory enters some terminal state thereafter executing a 1 cycle around this state.



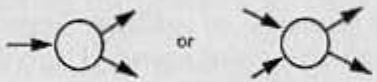
If the machine is state-determined (as so far supposed) then it is isolated (i.e. *Z* is invariant, *Ru* is invariant). Conversely an isolated system *is* such a machine, given agreement with postulates 1 to 5.

1.3 A collection of machines (of this type) is also a machine of this type; trivially, but correctly, if the machines are unconnected. This is so since it is always possible to refer to a state of the entire machine as the product of the states of the smaller machines. Suppose there is a limit upon storage (in the sense of an amount of order that can be accommodated in a unit of fabric), it would be possible to ensure that any physical machine is finite. An

appropriate limiting principle is furnished by Bremmerman's limit (Bremmerman, 1962; Ashby, 1968). Take any physical universe you like, where postulates 1 to 5 apply. It is, by the definition of 'physical universe', isolated and, by its nature, subject to Bremmerman's limit; hence, it is, if these postulates apply, a machine of this type, or a bag of them which is itself a machine of this type. In particular, any real (engineer's) machine is a machine of this (abstract) type.

It is worth noting that though I respect the utility of postulates 1 to 5 and the elegance of their development, I do not regard these principles as generally applicable or 'true'; nor do I concur in the philosophical conclusion about the universe or isolated bits of it. However, it is quite important to stress that my disagreement is not based on grounds directly related to the next (probablistic) development of machines.

1.4 Suppose that *Ru* is redefined as a relation rather than a function, so that pictures like the ones shown become permissible. Call the revised rule



RR. It is possible to characterise a new kind of machine (preserving all of the other requirements) by a mapping

$$RR: Z \rightarrow Z$$

which may be one to many. If so, the new kind of machine in state z_a (say, at τ) may go into state z_b or z_c at the next instant $\tau + 1$. But it must be in one and only one state at once. Since *RR* fails to prescribe which state this will be, and since it will be just one of the possibilities, it is usual to invoke the notion of chance or probability, i.e. to conceive of a random device that determines the outcome.

It is usually the case, and often harmless, to think of the random device that picks out labels for each next state permitted by *RR* as a wheel of chance, or, considered in a more sophisticated way, an uncorrelated source of events like a radioactively decaying material.

Whichever interpretation you choose, the following points are valid.

(a) Given a state at τ (say $z_\tau = z_a$) and given *RR*, the set of next states (up to all of them) permitted by *RR* is a set of exclusive and exhaustive alternatives.

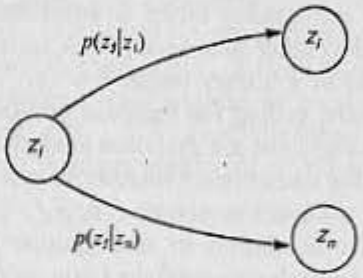
(b) Suppose this set is $\{z_b, z_c\}$ then one of $\{z_b, z_c\}$ is selected as $z_{\tau+1}$, but only one.

(c) An unbiassed random source would have the property that for any value of τ, if z_a occurs, then in the long run (over many occurrences of

$z_\tau = z_a$) on half the occasions the selected state would be z_b and on half z_c; a fact expressed by assigning a positive fractional number, p, to the states z_a, z_b with the property that the p numbers sum to unity and stating '$p(z_a) = p(z_b) = \frac{1}{2}$'. In the general case of m alternatives named by the value of an index j, then $p(z_j) = 1/m$. In so far as this metrical assertion tallies with the frequency count, as the number of instances is increased, $p(z_j)$ is the probability of z_j (and remains so, even if $p(z_j)$ departs from the equiprobability condition that $p(z_j) = 1/m$ for all values of j).

(d) Further, under the same restrictions, the conditional probability $p(z_j|z_i)$ is the probability, given the machine is in state z_i at τ, that it will be in state z_j at instant $\tau + 1$.

(e) As a refinement in machine design, it is possible to assert, for each z_i in Z a conditional probability $p(z_j|z_i)$ that a certain transition will occur. This trick amounts to using a systematically biassed random source or to biassing unbiassed random sources. Any machine of this kind is called probabilistic and is represented graphically as shown below



or, in so far as Z and *RR* are invariant, by a matrix with rows corresponding to all possibles states (those in Z) and columns corresponding to the same states (one instant later) and with entries (typified by the cell $\langle z_i, z_j \rangle$ with entry $p(z_j|z_i)$) that are conditional probabilities. If these entries do not change in value the machine is a Markovian device. Two representations are convenient. On the one hand *RR* can be stated together with a list, for each state z_i, of the probability numbers displayed in the graph of the machine. Call the list *Prob*: the machine is prescribed by

$$\langle Prob, RR \rangle : Z \rightarrow Z$$

Equally, if all of the caveats hold good, the conditional probability *matrix* represents the machine on its own.

(f) These are prescriptive expedients. There are also descriptive expedients, i.e. given an unknown but dynamic entity (which it is possible to isolate as an observable in some manner independent of the observation) its state

transitions can be observed. Such a machine is generally called a stochastic source rather than a probabilistic machine.

(g) If neither *Prob* nor *RR* change, the source is a *stationary* source (as the term is used in Chapter 1).

(h) Commonly, for reasons of accessibility, it is only possible to observe behaviours $\langle v_0, \ldots, v_\tau \rangle$ and not state trajectories $\langle z_0, \ldots, z_\tau \rangle$. If so, the source is incompletely observable.

(i) On examining any two or more sources, an observer, by virtue of having separated them, believes himself able to isolate them. This commits him to a tentative hypothesis that the machines are isolated; they are isolated to the best of his knowledge.

More exactly, using his observational language he has described state sets that are, according to his prevailing beliefs in the matter, distinct universes U_α and U_β (as in Chapter 1).

On the presupposition of isolation he can base an hypothesis that the machines might be related *beyond his knowledge*, that is, there might be a statistical dependency. Let the machines (alias sources) be called α and β. Let the observer use contingency tables or some comparable method to record event frequencies (either state occurrences, state transitions, outputs, behavioural sequences of arbitrary length $\tilde{v} = \langle v_1, \ldots, v_\tau \rangle$ for each of the machines separately, calling the recorded entities r^α (from α) and r^β (from β) and let him carry out a correlation analysis, which is a standard statistical technique for determining whether or not joint events such as the event $\langle r_\tau^\alpha, r_\tau^\beta \rangle$ or the event sequences $\langle \langle r_0^\alpha, r_0^\beta \rangle \langle r_1^\alpha, r_1^\beta \rangle, \ldots, \langle r_{\tau-1}^\alpha, r_{\tau-1}^\beta \rangle, \langle r_\tau^\alpha, r_\tau^\beta \rangle \rangle$ have a relation to one another (the *magnitude* and *sense* (+ or −) of the correlation coefficient says more than this).

(j) If, on this criterion, α and β have no relation then they are statistically independent. Computing the (*raw*) correlation as a transmission, we obtain the statement of Chapter 1

Independence if and only if $T(\alpha,\beta) = 0$

dependence if $T(\alpha,\beta) > 0$

1.5 Suppose a machine is not independent but receives an input, at synchronous instants, τ (or that it inspects an input *state* at these instants). Input states are exclusive and exhaustive alternatives like internal states; in particular there is one and only one input state at any instant. Amongst the inputs there is a value *null* that does nothing so that 'no input' and 'input is null in value' are synonymous. The input state at instant τ is designated u_τ. The set of inputs is U.

To accommodate an 'input' or external condition that acts upon the machine, *Ru* must be redefined, as follows, to specify a finite-state machine (FSM) of which the deterministic machine is a special case; namely

$$\text{FSM} \begin{Bmatrix} F\colon U \times Z \to Z \\ G\colon U \times V \to V \end{Bmatrix} \text{ together with an initial state } z_0 \in Z$$

or, writing these mappings as functions of two variables indexing U and Z

$$\text{FSM} \begin{Bmatrix} z_{\tau+1} = f(u_\tau, z_\tau) \\ v_{\tau+1} = f(u_\tau, z_\tau) \end{Bmatrix} \text{ together with an initial state } z_0 \in Z$$

The alternative forms, derived from *Ru1* rather than *Ru* is

$$\text{FSM} \begin{Bmatrix} F\colon U \times Z \times \mathrm{T} \to Z \\ G\colon U \times Z \times \mathrm{T} \to V \end{Bmatrix}$$

$$\text{or} \qquad \text{FSM} \begin{Bmatrix} z_{\tau+1} = f(u,z,\tau) \\ v_{\tau+1} = g(u,z,\tau). \end{Bmatrix}$$

The rules (the next internal state rule F, and the next output rule G) are still functions, but since they are functions of two variables (indexing the internal and the input states) it is quite possible for the machine to undergo a transition from one internal state into several next internal states depending upon the value of its input. It is convenient to represent all this by a state graph and to comment that (since the inputs form a state set) the inputs can be tagged by symbols, {null, *a*,*b* . . .} from an alphabet and outputs (since they also belong to a state set) can be tagged by symbols from a different alphabet {null, *A*,*B* . . .}. To avoid confusion, we omit *null* outputs completely in labelling machine graphs, of which the following example (Fig. 11) is typical.

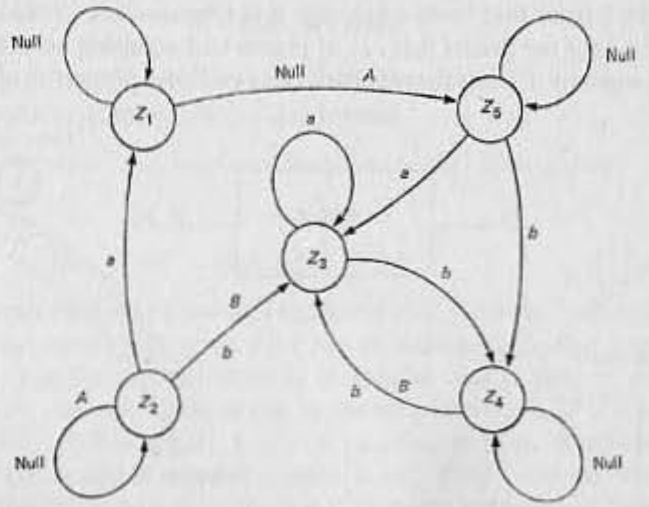


Figure 11 Typical state graph of a machine: internal states, $\{Z_1, Z_2, \ldots\}$; input states, (*null*, *a*, *b*, . . .}; output states, {*A*, *B*, . . .}. (Note: to avoid confusion null output states are not labelled.)

If more than one arc emerges from a node (representing an internal state) then these arcs must bear different labels. A machine of this kind is open, in a very real sense, to its input which is sometimes imaged as a tape (advanced by one step for each clock impulse) and sometimes as an environment.

The degree of freedom is considerable in view of the fact that we might just as readily regard the input as changing the rules of operation. That is, instead of writing some gigantic function (F,G) as a function of two variables it might have been supposed that F (say) is made up from different rules F_u and that u selects amongst them. This replaces

$$z_{\tau+1} = F(z_\tau, u_\tau)$$

by $$z_{\tau+1} = F_{u_\tau}(z_\tau)$$

However, there are restrictions. It is essential that the internal state set (or, as will be seen later, the set of irredundant internal states) does not change and the same applies to the input state set.

Further, inputs are strictly clocked and, being states, occur one and only one at a time and are unique. This requirement is usually represented by an input tape (see Fig. 12) read by the machine, bearing one state label on each segment and advanced one step for each τ. Similar comments apply to the output tape (except that its synchronisation is guaranteed by the machine which controls it and prints onto it). Ratchet wheel X moves for each state transition (on each output if the null state is counted). Ratchet wheel Y is moved by the τ clock. Without the arrangement or something equivalent the FSM must decode the actual and possibly asynchronical input into a form that looks as though it is synchronised (conceivably by regarding only the orders 0,1, . . . of inputs and counting an event as an input if, and only if, it has the exhaustive and exclusive properties of a state).

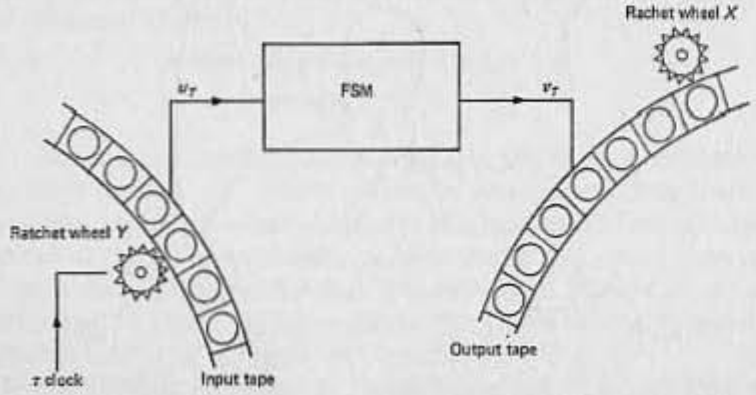


Figure 12 Machine with input and output tapes. Each tape segment bears a state label.

1.6 Just as there are probabilistic versions of machines with rule *Ru* (namely machines with rule *RR*) there are also probabilistic finite state machines obtained by essentially the same construction. It is quite important to notice *how* they are probabilistic.

There is a state transition matrix. But the entries in this matrix may depend upon the input state.

Thus: Probability of state at $\tau + 1$ = $Matrix_{u_\tau}$ (State at τ)

The inputs are not probabilities, and as before, the internal state set is unchanged, either by the stochastic operation of the device or by inputs applied to it (the converse construction is equivalent; namely, a deterministic automaton, and a probability vector input).

The most succinct and intelligible account of probabilistic automata is an article (on stochastic computation) by Gaines (1969a). The closely related problems of combating the 'noise' or 'random perturbations' that convert the computing elements of a real machine into devices that act like probabilistic automata, rather than finite deterministic automata, are clearly and elegantly discussed in Chapter 3 of Arbib (1964), which covers both the Von Neumann theory of 'redundant', or multiplexed, computation as a means of overcoming 'noise' and the Cowan–Winograd theory.

1.7 Other representations of machines are possible, for example, a listing of ordered sets of states shown as n-tuples. Thus, *Ru* is given by a list of pairs like

⟨state, next state⟩

or $\langle z, Ru(z)\rangle$

Similarly *RR* is given by a list of quadruples

⟨internal state, input state, output state, next internal state⟩

or $\langle z, u, v, \text{next } z\rangle$

or $\langle z, u, g(z,u), f(z,u)\rangle$

All these representations use the convenient idea of a state. The problem is that a large price must be paid for the convenience; for any reasonably sized machine the representation is impossible and is anyway generally unnecessary since the machine can be described in terms of 'components' or submachines into which it can be partitioned. Any 'component', of course, must compute a definite function and have a known relation to other components. But, given these requirements, a figure like Fig. 13 can be meaningfully partitioned into components that are FSMs interconnected together (Fig. 14). The picture is particularly neat if the unitary FSMs

inside the large FSM belong to only a few categories (for example, see the discussion, by Minsky (1967), concerning machines using McCulloch/Pitts elements as the units). The state of the FSM (if it is an FSM and is properly constructed/partitioned) is retrievable as the product of the states of the units.

Figure 13 Simple representation of a finite-state machine.

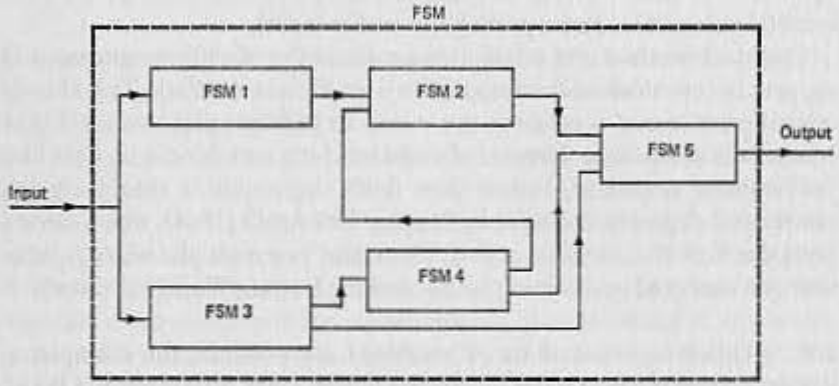


Figure 14 Example of partitioning a finite-state machine into components that are interconnected finite-state machines.

Such figures are simpler to manipulate, for all practical purposes, than state representations. The main justification for talking, improvidently, of states is because a state representation is more susceptible to logical analysis. One way of putting the matter is to note that an FSM can be regarded, in its state representation, as the 'grammar' or 'set of rewriting rules' (list of quadruples) and a formal language. Legal expressions in this formal language are read (symbol to input state) from the machine input tape and are printed (symbol to output state) on to the machine output tape. The internal states are non-terminal symbols in the language. These statements are couched in the terminology for that part of psycholinguistics concerned with phrase structure, grammars and the like. The crucial issue is the existence of a state-to-symbol correspondence. Because of this, particular machines can be characterised as able to 'accept' or to 'generate' certain classes of strings (the expression is 'grammatical' according to the grammar for the particular class of machine). Moreover, different classes of machines are able to compute, i.e. accept or generate strings, only of a certain type.

For example, FSMs, as a class, compute only *regular* expressions (Kleene's theorem).

Because that is so, the otherwise arbitrary idea of an output gains substance. A non-trivial output occurs if, and only if, a machine has recognised (in other words accepted) a string or expression belonging to some class; in which case it is a recogniser of that class of expressions. Further, the irredundant internal states of an FSM are not just mechanically ordained configurations but correspond to histories, i.e. finite expressions, that the FSM is able to recognise.

The serious reader should refer to Minsky (1967). Chapters 1 to 4 contain the most suitable treatment of the concepts I have just mentioned. The development of machine theory in the field of psycholinguistics is brilliant but has questionable relevance.[1] One compact account of machines as grammars and of the expressions they can generate or accept is in a series of papers by Chomsky and Miller (in Luce, Bush and Gallanter, 1963). It is well worth reading these articles and also the shorter account in Bach (1964) of what expressions can be produced/accepted by what machines.

1.8 One very important extension of finite state machines, Von Foerster's (1971) class of finite-function machines, is readily expressed using the notations of Figures 13 and 14 together with two additional symbols; namely '$\diagup$' to stand for a parametric operation upon the internal state set of a machine and '$\otimes$' to stand for a description of the condition of any machine that is operated upon parametrically (since, in the present case, the machine is a finite state machine, a description can be a record of its input/output history). A finite function machine is shown in Fig. 15

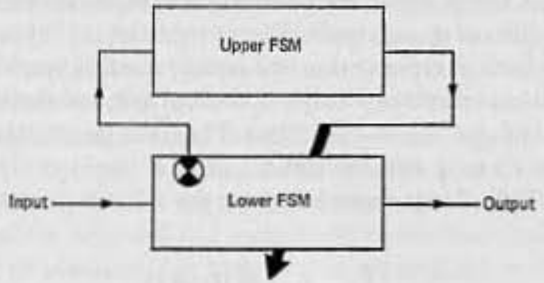


Figure 15 A finite-function machine.

1. There is no doubt about the validity or elegance of this work but I am inclined to the view that knowledge of the syntactic forms admissible for certain machines has little bearing upon the meaningful utterances made either by human beings or by machines. The reader should judge the matter for himself.

and derives its name from the fact (the irredundant internal state theorem) that all irredundant or relevant internal states can be represented as recursively expanded sequences of inputs and outputs, i.e. as state-history functions. The upper machine in Fig. 15, which takes these values as an input and computes an operation that transforms the internal state set (history function) of the lower machine is a function class or *functional*.

We can add comments to this. (a) Certain finite function machines reconstruct themselves in the way that has been described; they correspond to a subset of the self-reproducing automata to be introduced in section 1.10. (b) There is no reason why the lower box of Fig. 15 should not contain many, accumulated, finite-state machines generated, as variants, by this method, though, under the restrictions currently in force, they are synchronous and equivalent to a very large machine built up in stages.

1.9 A finite state machine is 'open' in so far as it does not control its environment. Unless the environment is already expressed as a tape with unique symbols inscribed on it, moved one step for each τ, it must decode the environment so that any input satisfies these constraints.

Now consider a finite-state machine that *can* control its environment, constrained as before, by moving in it or by printing symbols onto it. This environment is a series of linearly arranged storage locations, the 'machine tape', which is used for input and output and to which the machine has access by reading or printing one symbol at any one instant. The tape(s) may have an indefinite length(s); no size or storage limit is imposed. It is decreed, however, there is only one FSM in the environment, or if there are several, in practice they are coalesced to act synchronously as one FSM. On the other hand it does not matter a great deal in the abstract how many tapes there are. For example, the FSM might have an input and an output tape or a collection of storage tapes. These properties are important because they are used both in representing any serially clocked synchronous computer as an abstract machine which is equivalent to it, and also in developing the theory of self-reproductive machines. The entire (abstract) contraption; the FSM and its tape environment taken as a whole entity is a *Turing Machine*, or TM, if it is represented by the following set of specifying functions

$$\text{TM} \begin{Bmatrix} z_{\tau+1} = f(z_\tau, u_\tau) \\ v_{\tau+1} = g(z_\tau, u_\tau) \\ \text{Move}_{\tau+1} = h(z_\tau, u_\tau) \end{Bmatrix}$$

where 'move' means 'move on tape' or 'move tape' and has the possible values 'right' and 'left' and *null* or '+ 1 step' and '− 1 step', and *null*. The TM can also be represented by a list of ordered states or symbols standing

for them, for example quintuples like

$$\langle z_\tau, u_\tau, f(u_\tau, z_\tau), g(u_\tau, z_\tau), h(u_\tau, z_\tau) \rangle$$

or $$\langle z_\tau, u_\tau, z_{\tau+1}, v_{\tau+1}, \text{Move}_{\tau+1} \rangle$$

But a TM is represented more conveniently by a picture (Fig. 16). The ratchet-like object shown there is a device either for moving the tape or the locus of the reading input and the writing output on the tape.

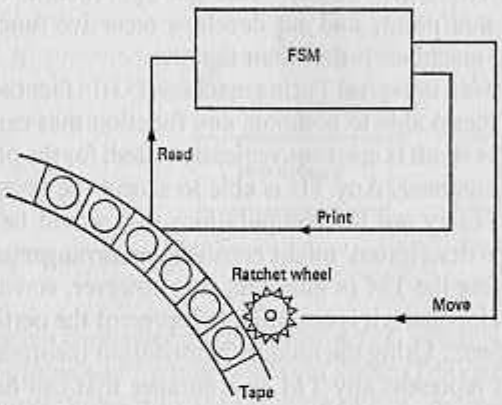


Figure 16 A Turing machine.

1.10 At this point, the serious reader should refer to Minsky (1967), Chapters 5 to 8, and Chapter 10; the other chapters are fascinating but optional. If he thinks, as I do, like a philosophically minded mechanic, this is by far the best and most lucid introduction, but it is also necessary to take up the references to Davis (1958) and it is an advantage (at least for people with an aptitude for pure and symbolic argument) to read Davis in his entirety. Perhaps this is a project to be undertaken at leisure; it appears on my postgraduate's reading list as a 'book to be mulled over'. But the mandate is crucial because the next few sections do nothing more than assert some results and properties. The detailed proofs of these results are relatively unimportant to the arguments in the sequel although they can be appreciated as beautiful and memorised by mathematicians. But it is illuminating to construct (on paper or as an artifact) entities with the required properties; for example, some finite-state machines and at least one universal Turing machine of your own design.

(a) Regarded as a grammar, a Turing machine permits the generation or acceptance of more expressions than the regular expressions of a finite-state machine. Regarded as a productive device a TM is an unrestricted rewriting device.

(b) Similar comments apply to a TM *qua* computing machine and its greater power may be ascribed to its greater storage capacity; for example an FSM cannot multiply arbitrary numbers. But of course, it is possible to simulate a finitary form of a TM in an FSM, capable of multiplying specified numbers. 'Add', for example, only entails the degree of (one stage) retention needed to realise a 'carry' operation. 'Multiply' may involve any number of 'retention stages'. In particular, there is a TM able to compute any general recursive function; Davis (1958) bases his approach to recursive computation upon this point, and he develops recursive function theory in terms of Turing machines rather than the reverse.

(c) Next, there is a universal Turing machine TU (in fact there are indefinite numbers of them) able to compute any function that can be computed by some TM. The result is most conveniently stated, for the present purpose, in the following manner. Any TU is able to accept the tape description of any TM and to carry out the computations that would be performed by this TM. A 'tape description' might consist in an arrangement of the quintuples representing the TM in question. It is however, convenient, and for subsequent developments it is essential, to represent the pertinent TM in an arithmeticised form. Using the unique factorisation theorem of arithmetic, it is possible to represent any TM as a number that can be unpacked by successive factoring to uniquely specify the TM. This numbering trick is essentially the trick employed by Godel in proving his famous theorem;[2] the number in question is often called the code number of the TM.

(d) Without *logical* innovation it is possible to specify a constructing machine which, given the tape description of any machine *TM*, call it *Descr* (*TM*), will produce a (chain-like, or tape-like) copy of *TM*. Further, the constructive automaton (let us call it *Build*) may form part of a larger automaton in which its operations are started and to which their completion is signalled. That is

$$Build\,(Descr\,(TM)) \rightarrow TM$$

(the original description being demolished in the process).

Next, there is, from the existence of *TU*, a further machine (called *Copy* as a tag word) that will copy any description such as *Descr* (*TM*) to produce copies of it. That is

$$Copy\,(Descr\,(TM)) = \langle Descr\,(TM),\ Descr\,(TM)\rangle$$

(the original description being demolished in the process).

2. In this respect Nagel and Neumann (1959) is an extremely readable book and provides a discussion, which is useful for the argument in the next volume, of the classical paradoxes and antinomies; notably Russell's paradox and the Richard paradox.

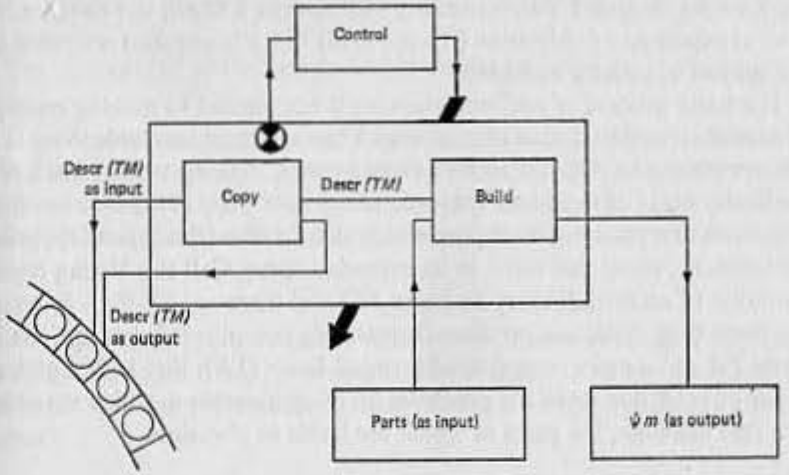


Figure 17 A universal constructor.

Finally there is a controller machine, also a *TM*, which is able to organise the activities of *Build* and *Copy* in respect of *Descr* (*TM*) which can institute the following operation (using the notation of Fig. 16, in Fig. 17).

$$Copy\,(Descr\,(TM)) \rightarrow \langle Descr\,(TM),\ Descr\,(TM)\rangle$$
$$Build\,(Descr\,(TM)) \rightarrow \langle TM,\ Descr\,(TM)\rangle$$

at the end of which *TM* and its description are cut free by punctuation. Call the entire machine *TR*; that is

$$TR = \langle Copy,\ Control,\ Build\rangle$$

so that $$TR\,(Descr\,(TM)) \rightarrow \langle TM,\ Descr\,(TM)\rangle$$

and *TR* is preserved.

There is a description of *TR*, *TM* and *TU*. Hence, it is possible to represent the process

$$TR\,(Descr\,(TR)) \rightarrow \langle TR,\ Descr\,(TR)\rangle$$

with the original *TR* being preserved.

This argument is a sketch of the demonstration given in Von Neumann (1966) lecture 5 and this reference (not only lecture 5 but all, at least, of part 1 of the book) should be consulted. The entirety *TR* is a Von Neumann 'self-reproducing' machine and it may be formulated in various ways that remove the 'linearity' (Von Neumann's word) of the tape descriptions *Descr* (*TM*). One of these ways, the Tesselation model (introduced in Part 2 of Von Neumann, 1966), is discussed briefly in the next chapter.

Burks, who compiled Von Neumann's lectures as a book, has contributed several papers to a publication (Burks, 1970) which is a crucial reference on the subject of cellular automata.

The basic process of self-reproduction is not limited to making replicas of a machine or population of machines. There are machines, able to execute the operations of *TR* and to do others as well. Amongst them there is a particular class of machines (Myhill, 1970; Loefgren, 1972) that produce sequences of replicating machines (which also do other things) and these are *evolutionary*, using this word in its everyday sense. Call the Turing representation of an evolutionary machine *TE*: and there are *TE* that, given all the parts they need, can produce increasingly complicated descriptions of novel *TR* and a more complicated form of *Descr* (*TE*). Fig. 18 is a picture either of evolution or of the preservation of an abstract machine structure in a real machine, the parts of which are liable to abrasion.

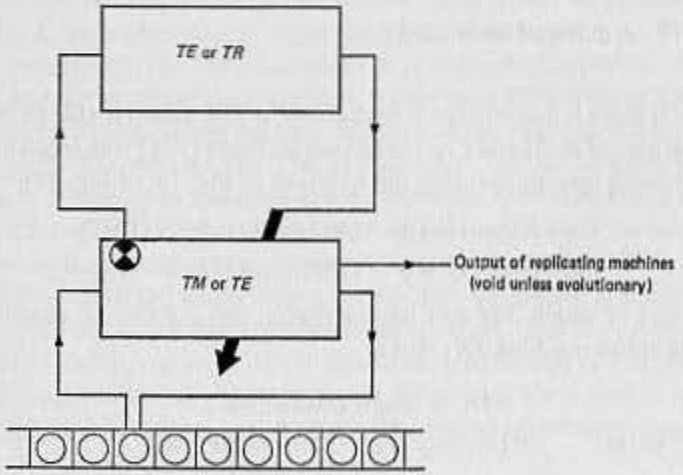


Figure 18 An evolutionary machine.

2 Fuzzy Abstract Machines

The theory of abstract machines is founded upon the concepts of set theory in so far as any such automaton has an internal state set, an input state set, and so on. In contrast, there exists a discipline known as fuzzy set theory, due to Zadeh (1968, 1971, 1973) and his colleagues.

The elements or members of a classical set definitely belong to it or not. The set is equivalently called a 'property'; any object either does or does not 'have' this property and if the property is *designated* by an adjective or predicate in a suitable language it may be *said* to have it or not. In contrast the elements or members of a *Fuzzy* set have a certain *grade of membership*.

This fact can be expressed numerically (as stated, very briefly, in Appendix C).

The concepts of immediate consequence are as follows.

2.1 An automaton with fuzzy input or state sets is a fuzzy automaton, and the computations it performs lead to grades of membership assignments. In general, the automaton computes a fuzzy relation; that is, it produces members of a (fuzzy) subset of the cartesian product of two or more fuzzy sets.

2.2 However, this automaton is strictly and serially clocked (its steps in operation are *not* fuzzy; the index set $t = 0,1, \ldots$ is not fuzzy, nor is the *index* set of the Cartesian product of which a fuzzy relation is a fuzzy subset).

2.3 In general, a step in the computation gives rise to a set of elements with varying grades of membership in each of the property values (alias sets) called the state variables. However, this *fuzzy outcome* can be resolved by various ranking operations involving the maximum values of the resulting cluster of grades of membership. If so, the operation of the automaton is said to be *numericised*, and the outcome at any step is resolved to yield a unique value, say the highest ranking according to a given scheme.

2.4 If the automaton is conceived as having one *state* (not one fuzzy state) at once, i.e. if it is strictly serial, then it can only operate in a numericised fashion. For example, a probabilistic automaton is a special case of a numericised fuzzy automaton.

2.5 If there are several independent copy automata; that is, automata running in parallel but still synchronously clocked by $t = 0,1, \ldots$ then those, in aggregate, constitute a fuzzy automaton and its operation need not be (though it *may* be) *numericised*.

3 Clocking and Programming

We should now recall the distinction, made right at the outset, between real machines and abstract machines or automata. Without qualification, the abstract machine mirrors the operation of an already *given* real machine. Moreover, a real machine can be designed using an abstract machine as a blueprint. But if so it is necessary to augment the blueprint by adding (apart from storage capabilities) a sequencing and clocking equipment to

make the real machine change state (to 'be executed'). For example, the machine can be furnished with a clock and a shift register, and these, as in section 1.1, correspond to abstract automata.

But, regardless of the representation used to depict it, the clock and the shift register *are* very definite entites indeed; those mooted in Chapter 1, section 10. The clock automation is executed by dint of a local supply of negentropy which converts a pattern (its design) into *information*. In turn, the clock controls the excursion of a local source of negentropy to scan an *ordered* (usually spatially ordered) storage medium and to convert inscribed patterns (as laid down by abstract design, or in other ways) into information; some of the information modifying (or addressing) the scanning locus. This arrangement (often abbreviated as 'clock') is a very primitive processor; a *real* not just an abstract, 'machine'. In general, the processor may consist in a number of clocks, driving counters and shift registers with ordered storage; these may or may not be synchronised. Finally, the storage may itself be dynamic, insofar as retention is accomplished by local sources of negentropy or by real machines that recompute patterns by retrieving information from them and converting it into further patterns at other addresses or positions in an ordering scheme. The existence of these more bizarre machine organisations is the main reason for the mild dissidence expressed in Section 1.2.

The notion of a processor is especially important if real (computing) machines are not so much *designed* as constructed by an *instruction* giving act, i.e. introducing a program and some data for it to operate upon as an initial state. The object into which the program is introduced is called a *processor* and it must have two basic capabilities: (a) interpretation; so that it can recognise and execute instructions; and (b) an ordering or clocking arrangement that allows it to execute interpreted instructions.

3.1 For a *serial* processor (Fig. 19(*a*)) a sequence of instructions, to set up the internal and input states of a real machine from an abstract machine described by the instructions, is an algorithm or program. Of several possible representations the most convenient is a series of conditional imperative statements ('If . . . then . . . else' statements) combined with the assignment statements (giving 'values to variables' or 'inscribing values on addressed location'). Thus a program is imaged by an ordered series of assignments and statements like

'If A then B else C'

For example ('If $x > y$ then $p \Rightarrow q$ else *to* statements n' or 'If $x = y$ then *to* statement m else *to* statement n'), in toto, computes a specific function.

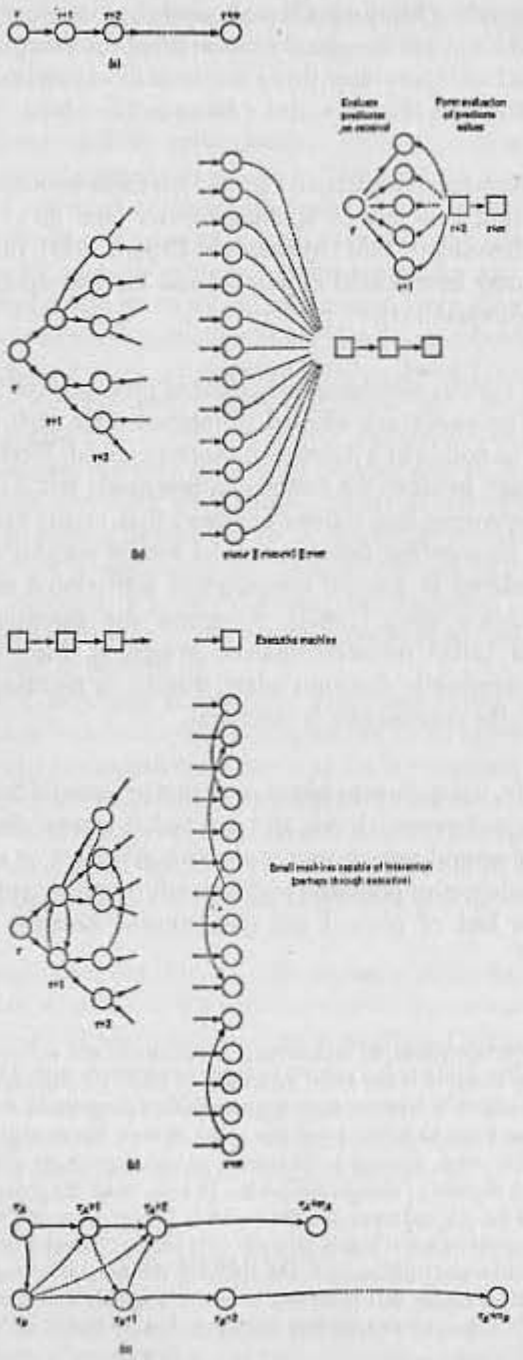
The corresponding fuzzy conditional imperative statements are outlined in Appendix C and are necessarily *numericised* for execution by a serial processor) and compute any fuzzy relation in numericised form. The program, in this case, is often called a *numerical heuristic*.

3.2 A processor is parallel (see Fig. 19(*b*)) if there are many synchronous clocks executing instructions simultaneously. One good example is a perceptron (Rosenblatt, 1961; Minsky and Papert, 1969). In this case fuzzy conditionals may be executed in parallel and the corresponding program is a *heuristic* (unqualified).

3.3 If the various simultaneously clocked processes (or loci of *control*) in a parallel processor are allowed to interact after each execution step (Fig. 19(*c*)), the notion of a *heuristic* is far from trivial. First, it is necessary to avoid conflict between the interacting processes; but it is also possible to obtain *cooperative* interactions provided that means are provided for *ordering* the transactions between several loci of control as they would have been ordered in a serial computation with stored results (see, for example, Simon's essay (1967)). Programs for execution by such a processor are called non-deterministic programs (Manna, 1970). The term 'non-deterministic' does not relate, directly or necessarily, to 'probabilistic'; here, the determinism is structural.

3.4 Finally, it is quite possible to construct processors that are provided with many asynchronous clocks, and in which it is possible to execute the operations of several asynchronous abstract machines or automata (Fig. 19(*d*)). This category of processors is inherently interesting since the initially asynchronous loci of control are synchronised because of *cooperative* interactions.[3]

3. Useful approximations to concurrent computation are achievable using serial processors. One example is the serial execution of SRETIC (Kilmer, McCulloch and Blum, 1969). Another is a recent language recognition programme due to Stavrinides. This programme takes as initial input the order of text; for example, of a children's story. It uses this order, and the juxtaposition of words (without other syntactic constraints) to form clusters of recognition nodes. In turn, these are grouped under further nodes. Whereas the original nodes are dealt with in a *given* order, the higher level nodes, to which they appear as novel inputs on a par with text, act 'simultaneously' in the sense that order is deliberately obfuscated. On applying the same construction heuristic to obtain higher level nodes that overlook both the originals and those nodes derived from them, it is necessary to establish discrimination by means of 'faster' processing (Stavrinides, 1973).



3.5 (a) The program for such a device is a fuzzy algorithm in which the index set (of well-ordered steps) is itself fuzzy; it is also a heuristic.

(b) The cooperative interactions between initially asynchronous loci of control, which synchronise them, are *information transfer* in the sense of Petri (1965) and Holt (1968), a concept quite distinct from the notion of information in Chapter 1, i.e. the interaction may be *cooperative* only if the asynchronous components become ordered by synchronisation (retrieving order, as in section 3.3).

(c) As was the case in section 3.3 as well, conflicting computations must either be avoided or tolerated as the price paid for a richer organisation.

Figure 19 Types of computation execution: (*a*) serial computation, from τ(start) to $\tau + m$(finish). At any point, locus of control may return to pick up stored data. Processor and program are one FSM and it has one state at once (the specification of which includes stored data). This FSM has no input; both initial state and computing inputs are read into processor, before execution starts, as program and data. (*b*) Parallel computation as a tree. Each locus of control '○' may be regarded as belonging to a small FSM but all of these FSMs are synchronous (so may be represented as one large FSM, i.e. their product); furthermore, they are independent and have no input. If the computation entails no conflict, the simultaneously submitted results are equivalent. Otherwise, as shown, they must be evaluated or assimilated by a further, serial FSM, with loci of control marked '□' to which the results are submitted as its input (signified by a dashed line). The loci of control '□' are at a higher level in an hierarchy of control than the '○' loci. Hence, a system with potential conflict is hierarchical. Minsky and Papert's analysis of a one layer perceptron is shown as a special and interesting case on the right. (*c*) Parallel computation with conflict-resolving interaction at each step (input to small FSMs corresponding to '○' loci of control from other small FSMs). If this cooperative interaction is organised by an executive machine at a higher level in hierarchy of control, input and output may take place through the executive (control loci '□') which prevents incompatible results and eliminates a need for an evaluating machine since all the FSMs are synchronous. It is possible to regard the entire processor and program as one large product FSM but, if so, the product (a peculiar one) has components at both levels of control in the system. (*d*) Concurrent computation by processor with initially asynchronous clocks (FSMs, loci of control) labelled $_{A, B}$ that become synchronised in conflict-resolving interactions, shown as dashed lines (these represent inputs, as before, but include generalised synchronising or 'change state' input). There are as many clocks as there are loci of control, though the effective number is reduced by partial synchronisation. The need to stipulate an hierarchy of control is eliminated but the operation of the system (in contrast to its 'physics') cannot be represented as a product machine with one, and only one, state at once. Also, there is not *necessarily* a *finish* to the process (we do not necessarily require that $\tau_A + m_A = \tau_B + m_B$); the same condition applies to *start*.