THREE

THE THEORY OF COMPUTATION

Thus far, we have discussed the limitations on computing imposed by the structure of logic gates. We now come on to address an issue that is far more fundamental: is there a limit to what we can, in principle, compute? It is easy to imagine that if we built a big enough computer, then it could compute anything we wanted it to. Is this true? Or are there some questions that it could never answer for us, however beautifully made it might be?

Ironically, it turns out that all this was discussed long before computers were built! Computer science, in a sense, existed before the computer. It was a very big topic for logicians and mathematicians in the thirties. There was a lot of ferment at court in those days about this very question — what can be computed *in principle*? Mathematicians were in the habit of playing a particular game, involving setting up mathematical systems of axioms and elements — like those of Euclid, for example — and seeing what they could deduce from them. An assumption that was routinely made was that any statement you might care to make in one of these mathematical languages could be proved or disproved, in principle. Mathematicians were used to struggling vainly with the proof of apparently quite simple statements — like Fermat's Last Theorem, or Goldbach's Conjecture — but always figured that, sooner or later, some smart guy would come along and figure them out[1]. However, the question eventually arose as to whether such statements, or others, might be inherently unprovable. The question became acute after the logician Kurt Gödel proved the astonishing result — in "Gödel's Theorem" — that arithmetic was incomplete.

3.1: Effective Procedures and Computability

The struggle to define what could and could not be proved, and what numbers could be calculated, led to the concept of what I will call an *effective procedure*. If you like, an effective procedure is a set of rules telling you, moment by moment, what to do to achieve a particular end; it is an algorithm. Let me

---

[1]In the case of Fermat's Last Theorem, some smart guy *did* come along and solve it! Fermat's Theorem, which states that the equation

$$x^n + y^n = z^n \ (n \text{ an integer}, n \geq 3)$$

has no solutions for which x, y and z are integers, has always been one of the outstanding problems of number theory. The proof, long believed impossible to derive (mathematical societies even offered rewards for it!), was finally arrived at in 1994 by the mathematicians Andrew Wiles and Richard Taylor, after many, many years' work (and after a false alarm in 1993). [Editors]

explain roughly what this means, by example. Suppose you wanted to calculate the exponential function of a number x, e^x. There is a very direct way of doing this: you use the Taylor series

$$e^x = 1 + x + (x^2/2!) + (x^3/3!) + ... \tag{3.1}$$

Plug in the value of x, add up the individual terms, and you have e^x. As the number of terms you include in your sum increases, the value you have for e^x gets closer to the actual value. So if the task you have set yourself is to compute e^x to a certain degree of accuracy, I can tell you how to do it — it might be slow and laborious, and there might be techniques which are more efficient, but we don't care: it works. It is an example of what I call an effective procedure.

Another example of an effective procedure in mathematics is the process of differentiation. It doesn't matter what function of a variable x I choose to give you, if you have learned the basic rules of differential calculus you can differentiate it. Things might get a little messy, but they are straightforward. This is in contrast to the inverse operation, integration. As you all know, integration is something of an art; for any given integrand, you might have to make a lot of guesses before you can integrate it: should I change variables? Do we have the derivative of a function divided by the function itself? Is integration by parts the way to go? In that we none of us have a hotline to the correct answer, it is fair to say that we do not possess an effective procedure for integration. However, this is not to say that such a procedure does not exist: one of the most interesting discoveries in this area of the past twenty years has been that there *is* such a procedure! Specifically, any integral which can be expressed in terms of a pre-defined list of elementary functions — sines, exponentials, error functions and so forth — can be evaluated by an effective procedure. This means, among other things, that machines can do integrals. We have to thank a guy named Risch for this ("The Problem of Integration in Finite Terms", *Trans. A.M.S.* 139(1969) pp. 167-189).

There are other examples in mathematics where we lack effective procedures; factoring general algebraic expressions, for example: there are effective procedures for expressions up to the fourth degree, but not fifth and over. An interesting example of a discipline in which every school kid would give his eye-teeth for an effective procedure is geometry. Geometrical proof, like integration, strikes most of us as more art than science, requiring considerable ingenuity. It is ironic that, like integration, there is an effective procedure for

geometry! It is, in fact, Cartesian analytic geometry. We label points by coordinates, (x,y), and we determine all lengths and angles by using Pythagoras' Theorem and various other formulae. Analytic geometry reduces the geometry of Euclid to a branch of algebra, at a level where effective procedures exist.

I have already pointed out that converting questions to effective procedures is pretty much equivalent to getting them into a form whereby computers can handle them, and this is one of the reasons why the topic has attracted so much attention of late (and why, for example, the notion of effective procedures in integration has only recently been addressed and solved). Now when mathematicians first addressed these problems, their interest was more general than the practical limits of computation; they were interested in principle with what could be proved. The question spawned a variety of approaches. Alan Turing, a British mathematician, equated the concept of "computability" with the ability of a certain type of machine to perform a computation. Church defined a system of logic and propositions and called it effective calculability. Kleene defined certain so-called "general recursive propositions" and worked in terms of these. Post had yet another approach (see the problem at the end of this chapter), and there were still other ways of examining the problem. All of these workers started off with a mathematical language of sorts and attempted to define a concept of "effective calculability" within that language. Thankfully for us, it can be shown that all of these apparently disparate approaches are equivalent, which means that we will only need to look at one of them. We choose the commonest method, that of Turing.

Turing's idea was to make a machine that was kind of an analogue of a mathematician who has to follow a set of rules. The idea is that the mathematician has a long strip of paper broken up into squares, in each of which he can write and read, one at a time. He looks at a square, and what he sees puts him in some state of mind which determines what he writes in the next square. So imagine the guy's brain having lots of different possible states which are mixed up and changed by looking at the strip of paper. After thinking along these lines and abstracting a bit, Turing came up with a kind of machine which is referred to as — surprise, surprise — a Turing machine. We will see that these machines are horribly inefficient and slow — so much so that no one would ever waste their time building one except for amusement — but that, if we are patient with them, they can do wonderful things.

Now Turing invented all manner of Turing machines, but he eventually discovered one — the so-called Universal Turing Machine (UTM) — which was the best of the bunch. Anything that any specific special-purpose Turing

machine could do, the UTM could do. But further, Turing asserted that *if anything could be done by an effective procedure, it could be done by his Universal machine, and vice versa*: if the UTM could not solve a problem, there was no effective procedure for that problem. Although just a conjecture, this belief about the UTM and effective procedures is widely held, and has received much theoretical support. No one has yet been able to design a machine that can outdo the UTM in principle. I will actually give you the plans for a UTM later. First, we will take a closer look at its simpler brother — the finite state machine.

3.2: Finite State Machines

A typical Turing machine consists of two parts; a tape, which must be of potentially unlimited size, and the machine itself, which moves over the tape and manipulates its contents. It would be a mistake to think that the tape is a minor addition to a very clever machine; without the tape, the machine is really quite dumb (try solving a complex integral in your head). We will begin our examination of Turing machines and what they can do by looking at a Turing machine without its tape; this is called a *finite state machine.*

Although we are chiefly interested in finite state machines (FSMs) as component parts of Turing machines, they are of some interest in their own right. What kinds of problems can such machines do, or not do? It turns out that there are some questions that FSMs cannot answer but that Turing machines can. Why this should be the case is naturally of interest to us. We will take all of our machines to be black boxes, whose inner mechanical workings are hidden from us; we have no interest in these details. We are only interested in their behavior. To familiarize you with the relevant concepts, let me give an example of a finite state machine (Fig. 3.1):

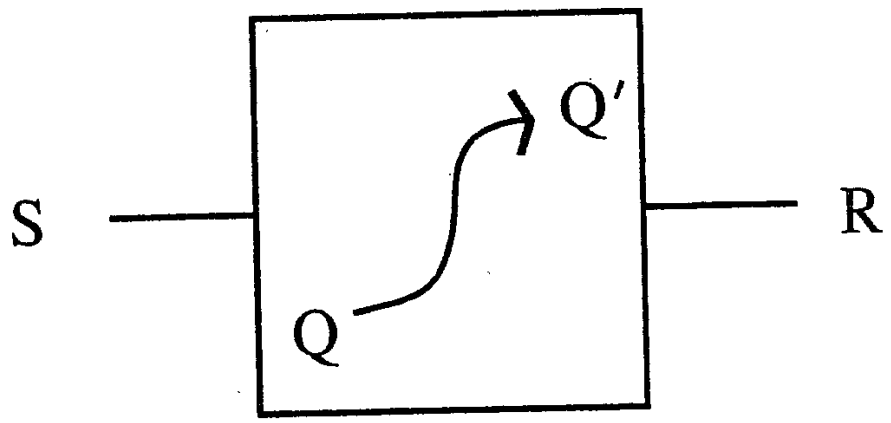


Fig. 3.1 A Generic Finite State Machine

The basic idea is as follows. The machine starts off in a certain *internal state*, Q. This might, for example, simply be holding a number in memory. It then receives an *input*, or *stimulus*, S — you can either imagine the machine reading a bit of information off a (finite) tape or having it fed in in some other way. The machine reacts to this input by *changing to another state*, Q', and spitting something out — a *response* to the input, R. The state it changes to and its response are determined by both the initial state and the input. The machine then repeats this cycle, reading another input, changing state again, and again issuing some response.

To make contact with real machines, we will introduce a discrete time variable, which sets the pace at which everything happens. At a given time t, we have the machine in a state $Q(t)$ receiving a symbol $S(t)$. We arrange things so that the response to this state of affairs comes one pulse later, at time $(t+1)$. Let us, for notational purposes, introduce two functions F and G, to describe the FSM and write:

$$\begin{aligned} R[t+1] &= F[S(t), Q(t)] \\ Q[t+1] &= G[S(t), Q(t)] \end{aligned} \tag{3.2}$$

We can depict the behaviour of FSMs in a neat diagrammatic way. Suppose a machine has a set of possible states $\{Q_j\}$. We represent the basic transition of this machine from a state Q_j to a state Q_k upon reception of a stimulus S, and resulting in a response R, as follows:

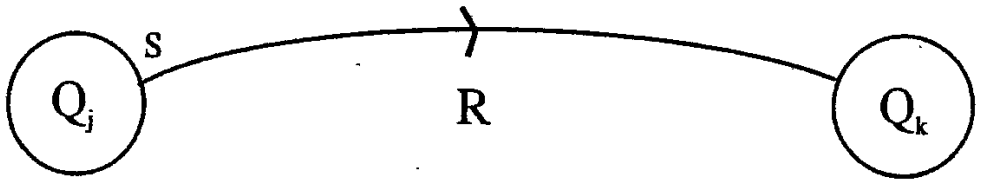


Fig. 3.2 A Graphical Depiction of a State Transition

This graphical technique comes into its own when we have the possibility of multiple stimuli, responses and state changes. For example, we might have the system shown below in Fig. 3.3:

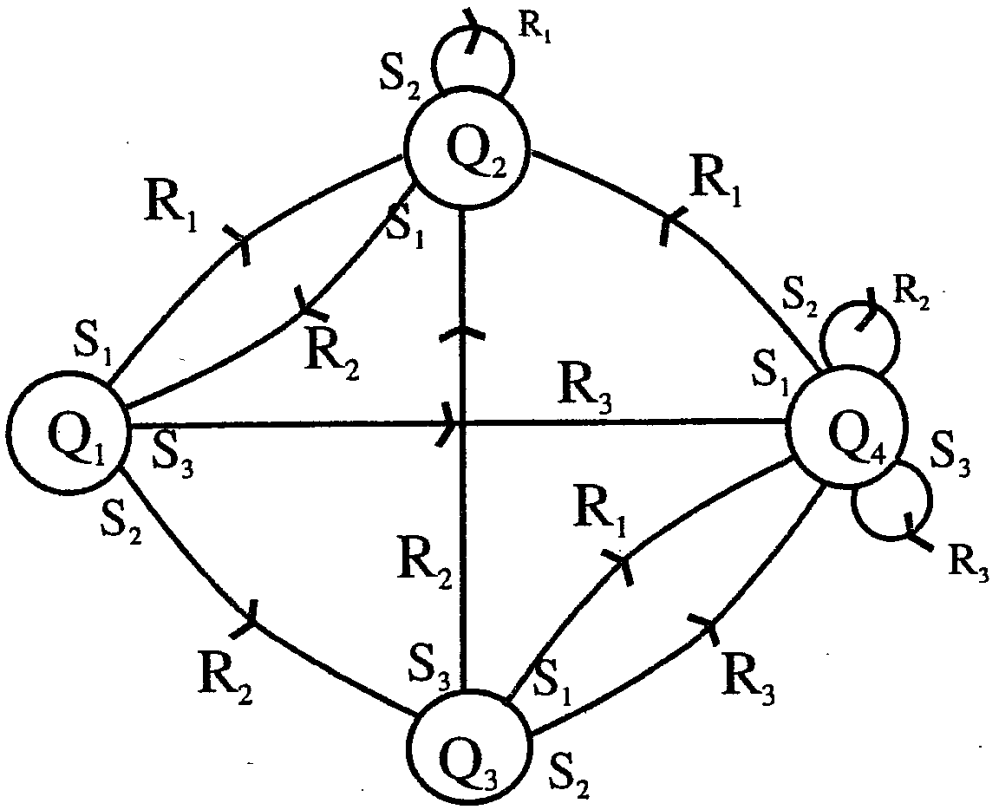


Fig. 3.3 A Complex Finite State Machine

This FSM behaves as follows: if it is in state Q_1 and it receives a stimulus S_1, it spits out R_1 and goes into state Q_2. If, however, it receives a stimulus S_2, it spits out R_2 and changes to state Q_3. Getting S_3, it switches to state Q_4 and produces R_3. Once in state Q_2, if it receives a stimulus S_1, it returns to state Q_1, responding with R_2, whilst if it receives a stimulus S_2 it stays where it is and spits out R_1. The reader can figure out what happens when the machine is in states Q_3 and Q_4, and construct more complex examples for himself.

One feature of this example is that the machine was able to react to three distinct stimuli. It will suit our purposes from here on to restrict the possible stimuli to just two — the binary one and zero. This doesn't actually affect what we can do with FSMs, only how quickly we can do it; we can allow for the possibility of multiple input stimuli by feeding in a sequence of 1's and 0's, which is clearly equivalent to feeding in an arbitrary number, only in binary format. Simplifications of this kind are common in the study of FSMs and Turing machines where we are not concerned with their speed of operation.

Let me now give a specific example of an FSM that actually does something, albeit something pretty trivial — a delay machine. You feed it a

stimulus and, after a pause, it responds with the same stimulus. That's all it does. Figure 3.4 shows the "state diagram" of such a delay machine.

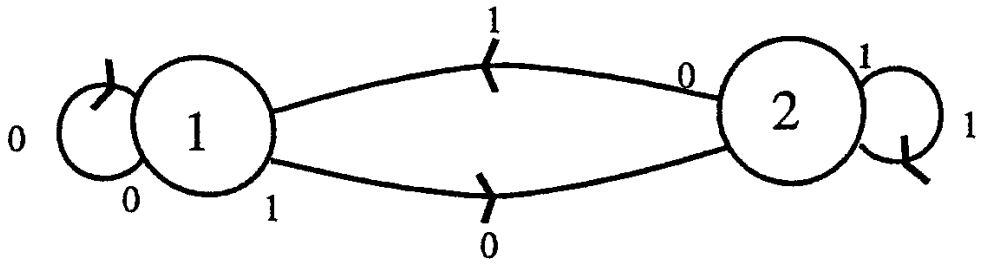


Fig. 3.4 A Delay Machine

You can hardly get a simpler machine than this! It has only two internal states, and acts as a delay machine solely because we are using pulsed time and demanding that the machine's response to a stimulus at time t comes at time t+1. If we tell our machine to spit out whatever we put in, we will have a delay time of one unit. It is possible to increase this delay time, but it requires more complicated machines. As an exercise, try to design a delay FSM that remembers *two* units into the past: the stimulus we put in at time t is fed back to us at time t+2. (Incidentally, there is a sense in which such a machine can be taken as having a memory of only one time unit: if we realize that the *state* at time t+1 tells us the input at time t. It is often convenient to examine the state of an FSM rather than its response.)

Another way of describing the operation of FSMs is by tabulating the functions F and G we described earlier. Understanding the operation of an FSM from such a table is harder than from its state diagram, and becomes hopeless for very complex machines, but we will include it for the sake of completeness:

G	Q_0	Q_1
S_0	Q_0	Q_0
S_1	Q_1	Q_1

F	Q_0	Q_1
S_0	R_0	R_1
S_1	R_0	R_1

Table 3.1 State Table for a Generic FSM

Now it is surprising what you can do with these things, and it is worth getting used to deciphering state diagrams so that you can appreciate this. I am going to give you a few more examples, a little more demanding than our delay machine. First up is a so-called "toggle" or "parity" machine. You feed this machine a string of 0's and 1's, and it keeps track of whether the number of 1's it has received is even or odd; that is, the parity of the string.

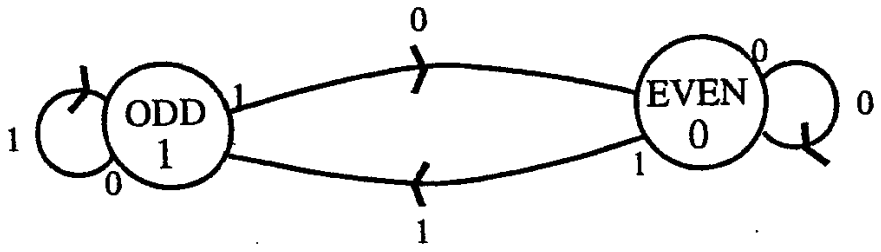


Fig. 3.5 The Parity Machine

From the diagram in Figure 3.5, you can see that, one unit of time after you feed in the last digit, the response of the FSM tells you the parity. If it is a 1, the parity is odd — you have fed in an odd number of 1's. A 0 tells you that you have fed in an even number. Note that, as an alternative, the parity can be read off from the state of the machine; which I have flagged by labeling the two possible states as "odd" and "even".

Let me give you some simple problems to think about.

Problem 3.1: Suppose we feed a sequence of 1's and 0's — a binary number — into a machine. Design a machine which performs a pairwise sum of the digits, that is, one which takes the incoming digits two at a time and adds them, spitting the result out in two steps. So, if two digits come in as 00, it spits out 00; a 10 or 01 results in a 01 (1+0 = 0+1!); but a 11 results in binary 10: 1+1 = 2, in decimal, 10 in binary. I will give you a hint: the machine will require four states.

Problem 3.2: Another question you might like to address is the design of another delay machine, but this time one which remembers and returns *two* input stimuli. You can see that such a device needs four states — corresponding to the four possible inputs 00, 01, 10 and 11.

Problem 3.3: Finally, if you are feeling particularly energetic, design a two-

input binary adder. I want the full works! I feed in two binary numbers, simultaneously, one bit from each at a time, *with the least significant bits first*, and the FSM, after some delay, feeds me the sum. I'm not interested in it telling me the carry, just the sum. We can schematically depict the desired behaviour of the machine as follows:

Time ⟶	
Inputs	1 0 1 0 1 1
	0 1 1 0 1 0
Output = sum	1 1 0 1 0 0 (Carrying 1 into the next column)

3.3: The Limitations of Finite State Machines

If you have succeeded in designing an adder, then you have created a little wonder — a simple machine that can add two numbers of any size. It is slow and inefficient, but it does its job. This is usually the case with FSMs. However, it is important to appreciate the limitations of such machines; specifically, there are many tasks that they cannot perform. It is interesting to take a look at what they are. For example, it turns out that one cannot build a FSM that will multiply any two arbitrary binary numbers together. The reason for this will become clear in just a moment, after we have examined a simpler example. Suppose we want to build a *parenthesis checker*. What is one of these? Imagine you have a stream of parentheses, as follows:

(((( )) ((( ) ( ) ((( )) ( ) ( ) (( ))) ( ))

The task of a parenthesis checker is to ascertain whether such an expression is "balanced": that the brackets open and close consistently throughout the expression. This is not the same as just counting the number of left and right brackets and making sure they are equal! They have to match in the correct order. This is a common problem in arithmetic and algebra, whenever we have operations nested within others. The above example, incidentally, is invalid; this one:

(( ) (( ) ( )) ((( ) ( ) (( ) ( )))))

is valid. You might like to check in each case.

On the face of it, building a parenthesis checker seems a pretty straightforward thing to do. In many ways it is, but anything you get to implement the check would not be an FSM. Here is one way you could proceed. Starting from the left of the string, you count open brackets until you reach a close bracket. You "cancel" the close bracket with the rightmost open bracket, then move one space to the right. If you hit a close bracket, cancel it with another open bracket; if you hit an open bracket, add one to the number of open brackets you have uncanceled and move onto the next one. It is a very simple mechanism, and it will tell you whether or not your parenthesis string is OK: if you have any brackets left over after you process the rightmost one, then your string is inconsistent. So why cannot an FSM do something this simple?

The answer is that the parenthesis checker we want has to cope with *arbitrary* strings. That means, in principle, strings of arbitrary length which might contain arbitrarily large numbers of "open" brackets. Now recall that an essential feature of the machine is that it must keep track of how many open brackets remain uncancelled by closed ones at each stage of its operation; yet to do this, in the terminology of FSMs, it will need a distinct state for each distinct number of open brackets. Here lies the problem. An *arbitrary* string requires a machine with an arbitrary — that is, ultimately, infinite — number of states. Hence, no *finite* state machine will do. What will do, as we shall see, is a Turing machine — for a Turing machine is, essentially, an FSM with infinite memory capacity.

For those who think I am nitpicking, it is important to reiterate that I am discussing matters of principle. From a practical viewpoint, we can clearly build a finite state machine to check the consistency of any bracket string we are likely to encounter. Once we have set its number of states, we can ensure that we only feed it strings of an acceptable size. If we label each of its states by 32 32-bit binary numbers we can enumerate over 2^{1000} states, and hence deal with strings 2^{1000} brackets long. This is far more than we are ever likely to encounter in practice: by comparison, note that current estimates place the number of protons in the universe only of the order of 2^{200}. But from a mathematical and theoretical standpoint, this is a very different thing from having a universal parenthesis checker: it is, of course, the difference between the finite and the infinite, and when we are discussing academic matters this is important. We *can* build an FSM to add together two arbitrarily large numbers; we *cannot* build a parenthesis checking FSM to check any string we choose. Incidentally, it is the need for an infinite memory that debars the construction of an FSM for binary multiplication.

Before getting onto Mr. Turing and his machines, I would like to say one or two more things about those with a finite number of states. One thing we looked at in detail in previous chapters was the extent to which complicated logic functions could be built out of simple, basic logic units — such as gates. A similar question arises here: is there a core set of FSMs with which all others can be built? To examine this question, we will need to examine the ways in which FSMs can be combined.

Figure 3.6 shows two machines, which I call **A** and **B**. I have linked them up in something of a crazy way, with feedback and whatnot. Don't worry if you can't see at a glance what is going on!

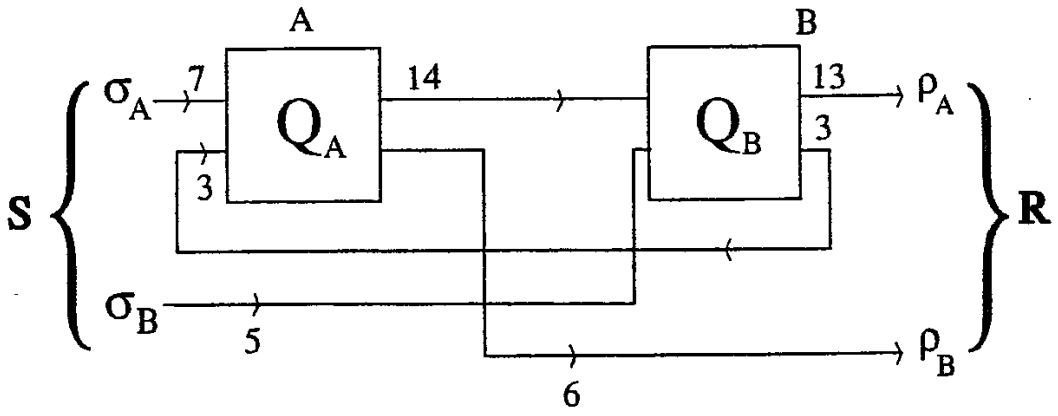
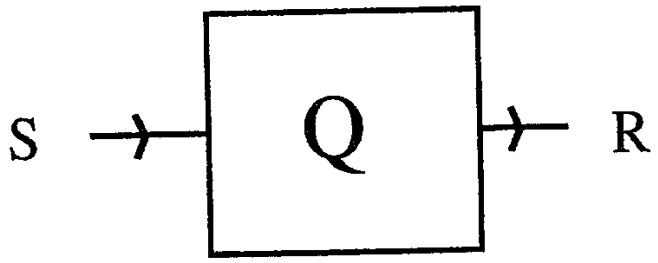


Fig. 3.6 A Composite FSM

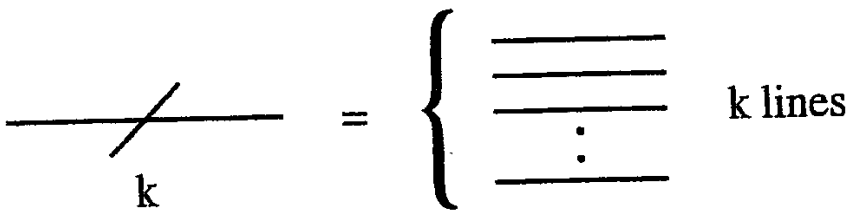
Let me describe what the diagram represents. In a general FSM, the input stimulus can be any binary number, as can its response. Whether the stimulus is fed in sequentially, or in parallel (e.g. on a lot of on/off lines), we can split it up into two sets. Suppose the stimulus for **A** has ten bits. We split this up into, say, a 7-bit and a 3-bit stimulus. Now comes the tricky part: we take the 7-bit input to be external, fed in from outside on wire σ_A, but the 3-bit input we take from the *response of machine B* — which we have also split up. In the case of **B**, we take the response to have, say, 16 bits, and 3 of these we re-route to **A**, the other 13 we take as output. Bear with me! What about the response of **A**? Again, we split this up: suppose it is 20 bits. We choose (this is all arbitrary) to feed 14 into **B** as input, and with the remaining 6 we bypass **B** and take them as output. The remainder of **B**'s input — whatever it may be — is fed in from outside on wire σ_B. Let's say σ_B carries 5 bits.

The point of all these shenanigans is that this composite system can be represented as a single finite state machine:



where the input stimulus is the combined binary input on wires σ_A and σ_B, and the output is the partial responses from **A** and **B**, again combined. Clearly, the machine has an input stimulus of 7+5=12 bits, and a response of 13+6=19 bits. Exactly what the thing does depends on the properties of **A** and **B**; it seems feasible that the number of internal states of this combined machine is the product of the number of states of **A** and **B**, but one must be careful about the extent to which things can be affected by feedback and the information running around the wires. What I wanted to show was how you could build an FSM from smaller ones by tying up the loose wires appropriately. You might like to see what happens if you arrange things differently – by forgetting feedback, for example. You will find that feedback is essential if you want as few constraints as possible on the size of the overall input and output bit sizes: connecting up two machines by, say, directly linking output to input not only fixes the sizes of the overall stimulus/response but also requires the component FSMs to match up in their respective outputs and inputs.

Let me return to my question: can we build any FSM out of a core set of basic FSMs? The answer turns out to be yes: in fact, we find ourselves going right back to our friends AND and NOT, which can be viewed as finite state machines themselves, and which we can actually use to build any other FSM. Let me show roughly how this is done. We will first need a bit of new notation. Let us represent a set of k signal-bearing wires by a single wire crossed with a slash, next to which we write the number k:



With this convenient diagrammatic tool, we can draw a schematic diagram of

a general finite state machine (Fig. 3.7):

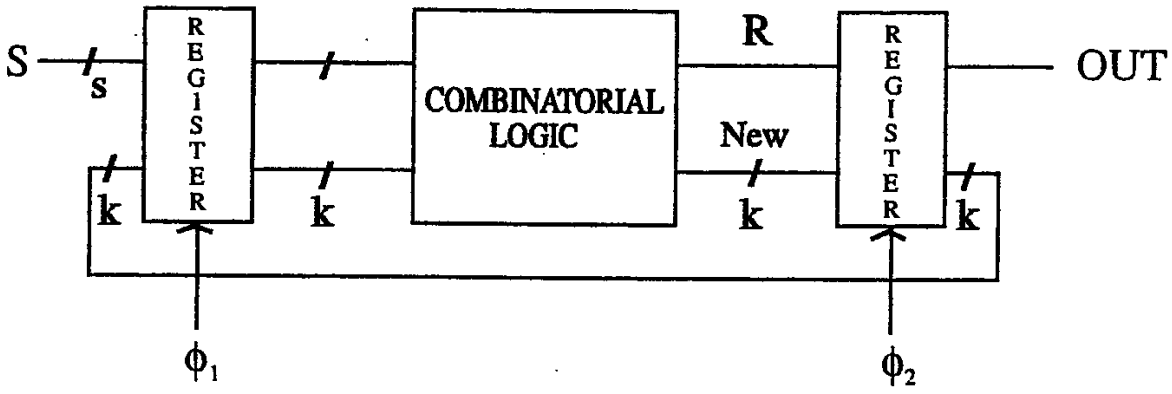


Fig. 3.7 The General FSM

The operation of this rather complicated-looking device is quite straightforward. It comprises two registers (such as those we constructed in Chapter 2 from clocked flip-flops) and a black box that performs certain logical functions. The input to the first register has two pieces, the stimulus S to the FSM and the state the machine is in, Q: central to our design is the fact that we can label the internal states by a binary number. In this case, the stimulus has s bits, and is fed in on s wires, while the state has k bits, fed in on k wires. (The FSM has therefore up to 2^k internal states). Subject to timing, which I will come back to, the register passes these two inputs into the logic unit. Here is the trick. An FSM, in response to a given stimulus and being in a given state, produces a response and goes into a (possibly) new state. In terms of our current description, this simply amounts to our black box receiving two binary strings as input, and producing two as output – one representing the response, the other the new state. The new state information is then fed back into the first register to prime the machine for its next stimulus. Ensuring that the FSM works is then just a matter of building a logic unit which gives the right outputs for each input, which we know is just a matter of combining ANDs and NOTs in the right way.

A quick word about timing. As we have discussed, the practicalities of circuit design mean that we have to clock the inputs and outputs of logic

devices; we have to allow for the various delays in signals arriving because of finite travel times. Our FSM is no exception, and we have to connect the component registers up to two clocks as usual; the way these work is essentially the same as with standard logic circuits. The first register is clocked by ϕ_1, the second by ϕ_2, and we arrange things such that when one is on, the other is off – which we do by letting ϕ_2 = NOT ϕ_1 and hooking both up to a standard clock – and ensuring that the length of time for which each is on is more than enough to let the signals on the wires settle down. The crucial thing is to ensure that ϕ_2 is *off* whilst ϕ_1 is *on*, to prevent the second register sending information about the change of state to the first while it is still processing the initial state information.

Problem 3.4: Before turning to Turing machines, I will introduce you to a nice FSM problem that you might like to think about. It is called the "Firing Squad" problem. We have an arbitrarily long line of identical finite state machines that I call "soldiers". Let us say there are N of them. At one end of the line is a "general", another FSM. Here is what happens. The general shouts *"Fire"*. The puzzle is to get all of the soldiers to fire simultaneously, in the shortest possible time, subject to the following constraints: firstly, time goes in units; secondly, the state of each FSM at time T+1 can only depend on the state of its next-door neighbors at time T; thirdly, the method you come up with must be independent of N, the number of soldiers. At the beginning, each FSM is quiescent. Then the general spits out a pulse, *"fire"*, and this acts as an input for the soldier immediately next to him. This soldier reacts in some way, enters a new state, and this in turn affects the soldier next to him, and so on down the line. All the soldiers interact in some way, yack yack yack, and at some point they become synchronized and spit out a pulse representing their "firing". (The general, incidentally, does nothing on his own initiative after starting things off.)

There are different ways of doing this, and the time between the general issuing his order and the soldiers firing is usually found to be between $3N$ and $8N$. It is possible to prove that the soldiers cannot fire earlier than T=$2N$-2 since there would not be enough time for all the required information to move around. Somebody has actually found a solution with this minimum time. That is very difficult though, and you should not be so ambitious. It is a nice problem, however, and I often spend time on airplanes trying to figure it out. I haven't cracked it yet.

3.4: Turing Machines

Finally, we come to Turing machines. Turing's idea was to conceive of himself, or any other mathematician, as a machine, having a finite state machine in his head, and an unlimited amount of paper at his disposal to write on. It is the unlimited paper — hence effectively unbounded memory — that distinguishes a Turing machine from an FSM. Remember that some problems — parenthesis checking, multiplication — cannot be done by finite state machines, because, by definition, they lack an unlimited memory capacity. This restriction does not apply to Turing machines. Note that we are not saying that the amount of paper attached to such a machine *is* infinite; at any given stage it will be finite, but we have the option of adding to the pile whenever we need more. Hence our use of the word "unlimited".

Turing machines can be described in many ways, but we will adopt the picture that is perhaps most common. We envisage a little machine, with a finite number of internal states, that moves over a length of tape. This tape is how we choose to arrange our paper. It is sectioned off into cells, in each of which might be found a symbol. The action of the machine is simple, and similar to that of an FSM: it starts off in a certain state, looking at the contents of a cell. Depending on the state, and the cell contents, it might erase the contents of the cell and write something new, or leave the cell as it is (to ensure uniformity of action, we view this as erasing the contents and writing them back in again). Whatever it does, it next moves one cell to the left or right, and changes to a new internal state. It might look something like Figure 3.8:

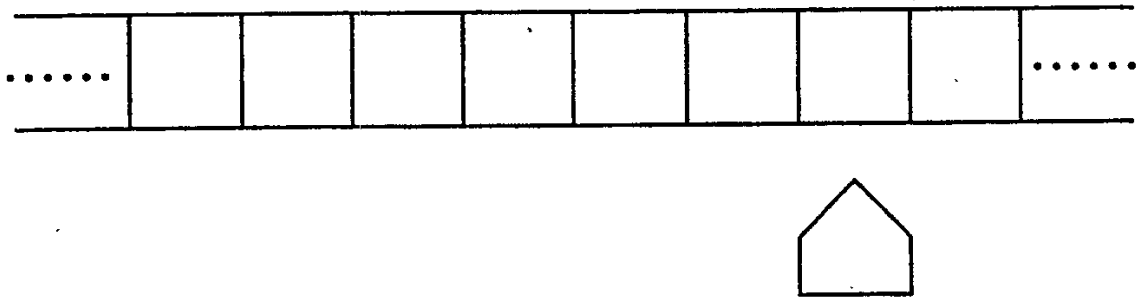


Fig. 3.8 A Turing Machine

We can see how similar the Turing machine is to an FSM. Like an FSM, it has internal states. Reading the contents of a cell is like a stimulus, and overwriting the contents is like a response, as is moving left or right. The restriction that the machine move only one square at a time is not essential; it just makes it more

primitive, which is what we want. One feature of a Turing machine that *is* essential is that it be able to move both left and right. You can show (although you might want to wait until you are more familiar with the ideas) that a Turing machine that can only move in one direction is just a finite state machine, with all its limitations.

Now we are going to start by insisting that only a finite part of the tape have any writing on it. On either side of this region, the tape is blank. We first tell the machine where to start, and this is at time T. Its later behavior, at a time $T+1$ say (Turing machines operate on pulsed time like FSMs), is specified by three functions, each of which depends on the state Q_i at time T and the symbol S_i it has just read: these are its new state, Q_j, the symbol it writes, S_j, and the direction of its subsequent motion, D. We can write:

$$\begin{aligned} Q_j &= F(Q_i, S_i) \\ S_j &= G(Q_i, S_i) \\ D &= D(Q_i, S_i) \end{aligned} \tag{3.3}$$

This list is just like the specification of an FSM but with the extra function D. The complete machine is fully described by these functions, which you can view as one giant (and finite) look-up list of "quintuples" — a fancy name for the set of five functions we have defined, two at time T (Q_i and S_i), and three at $T+1$ (Q_j, S_j and D). All you do now is stick in some data — which you do by writing on the tape and letting the machine look at it — tell the machine where to start, and leave it to get on with it. The idea is that the machine will finish up by printing the result of its calculation somewhere on the tape for you to peruse at your leisure. Note that for it to do this, you have to give it instructions as to when it is to halt or stop. This seems pretty trivial, but as we will see later, matters of "halting" hide some very important, and very profound, issues in computation.

Before giving you some concrete examples of Turing machines, let me remind you of why we are looking at them. I have said that finding an effective procedure for doing a problem is equivalent to finding a Turing machine that could solve it. This does not seem much of an insight until we realize that among the list of all Turing machines, by which I mean all lists of quintuples, there exists a very special kind, a *Universal Turing machine* (UTM), which can do anything any other Turing machine can do! Specifically, a UTM is an

imitator, mimicking the problem-solving activities of simpler Turing machines. (I say "a" UTM, rather than "the" UTM since, while all UTMs are computationally equivalent, they can be built in many different ways). Suppose we have a Turing machine, defined by some list of quintuples, which computes a particular output when we give it a particular set of input data. We get a UTM to imitate this process by feeding it a description of the Turing machine — that is, telling the UTM about the machine's quintuple list — and the input data, both of which we do by writing them on the UTM's tape in some language it understands, in the same way we feed data into any Turing machine. We also tell the UTM where each begins and ends[2]. The UTM's internal program then takes this information and mimics the action of the original machine. Eventually, it spits out the result of the calculation: that is, the output of the original Turing machine. What is impressive about a UTM is that *all* we have to do is give it a list of quintuples and some initial data — its own set of defining quintuples suffice for it to mimic any other machine. We don't have to change them for specific cases[3]. Why such machines are important to us is because it turns out that, if you try to get a UTM to impersonate *itself*, you end up discovering that there are some problems that no Turing machine — and hence no mathematician — can solve!

Let us now look at a few real Turing machines. The first, and one of the simplest, is related to a finite state machine we have already examined — a parity counter. We feed the machine a binary string and we want it to tell us whether the number of 1's in the string is odd or even. Schematically we have (Fig. 3.9):

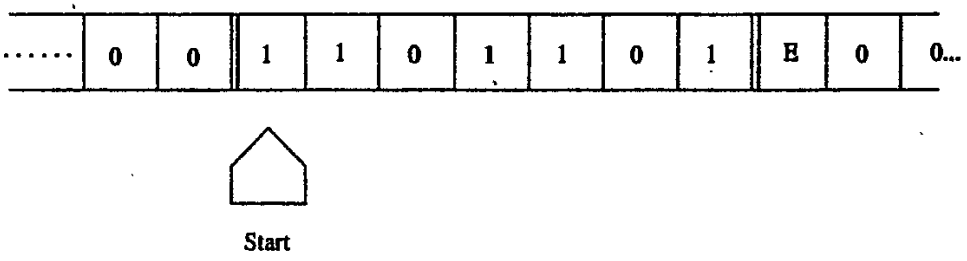


Fig. 3.9 Input Tape for the Parity Counter

We begin by writing the input data, the binary string, onto the tape as shown;

[2]The section of the UTM's tape containing information about the machine it is imitating is usually referred to as the "pseudotape". [RPF]

[3]We will actually construct a UTM later. [RPF]

each cell of the tape holds one digit. The "tape-head" of the machine rests at the far left of the string, on the first digit, and we define the machine to be in state Q_0. To the left of the string are nothing but zeroes, and to the right, more zeroes – although we separate these from the string with a letter E, for "end", so that the machine does not assume they are part of it.

The operation of the machine, which we will shortly translate into quintuples, is as follows. The state of the machine tells us the parity of the string. The machine starts off in state Q_0, equal to even parity, as it has not yet encountered any 1s. If it encounters a zero, it stays in state Q_0 and moves one space to the right. The state does not change because the parity does not change when it hits a zero. However, if it hits a 1, the machine erases it, replaces it with a zero, moves one space to the right, and enters a state Q_1. Now if it hits a zero, it stays in state Q_1 and moves a space to the right, as before. If it hits a 1, it erases it, putting a zero in its place, and moves to the right, this time reverting to state Q_0. You should now have an idea what is happening. The machine works its way across the string from left to right, changing its state whenever it encounters a 1, and leaving a string of 0s behind. If the machine is in state Q_0 when it kills the last digit of the string, then the string has even parity; if it is in state Q_1, it is odd. How does the machine tell us the parity? Simple – we include a rule telling the machine what to do if it reads an E. If it is in state Q_0 and reads E, it erases E and writes "0", meaning even parity. In state Q_1, it overwrites E with a "1", denoting odd parity. In both cases it then enters a new state Q_H, meaning "halt". It does not need to move to the right or left. We examine the tape, and the digit directly above the head is the answer to our question. We end up with the situation shown in Figure 3.10:

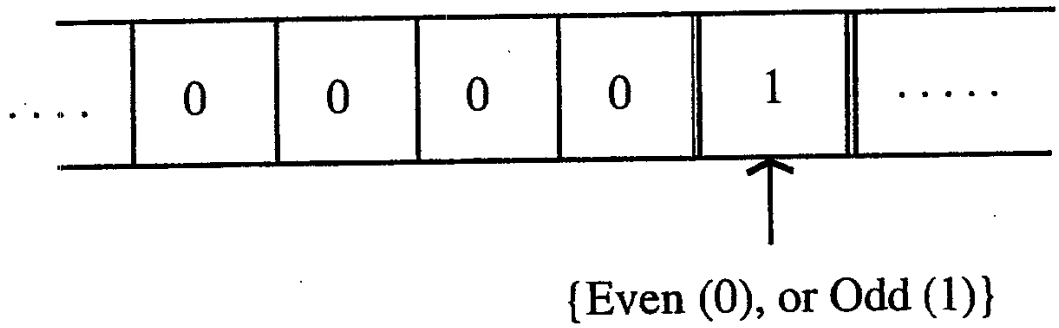


Fig. 3.10 Output Tape from the Parity Counter

The quintuples for this machine are straightforwardly written out (Table 3.2):

Initial State	Read	New State	Write	Direction of Move
0	0	0	0	R
0	1	1	0	R
1	0	1	0	R
1	1	0	0	R
0	E	H(alt)	0	-
1	E	H	1	-

Table 3.2 Quintuples for the Parity Couuter

Now this device is rather dumb, and we have already seen that we could solve the parity problem with a finite state machine (note here how our Turing machine has only moved in one direction!). We will shortly demonstrate the superiority of Mr. Turing's creations by building a parenthesis checker with them, something which we have seen cannot be done with an FSM, but first let me introduce some new diagrammatics which will make it easier for us to understand how these machines work without tying ourselves in knots wading through quintuple lists.

The idea is, unsurprisingly, similar to that we adopted with FSMs. In fact, the only real difference in the diagrams is that we have to somehow include the direction of motion of the head after it has overwritten a cell, and we have to build in start and halt conditions. In all other respects the diagrams resemble those for FSMs. Take a look at Figure 3.11, which describes our parity counter:

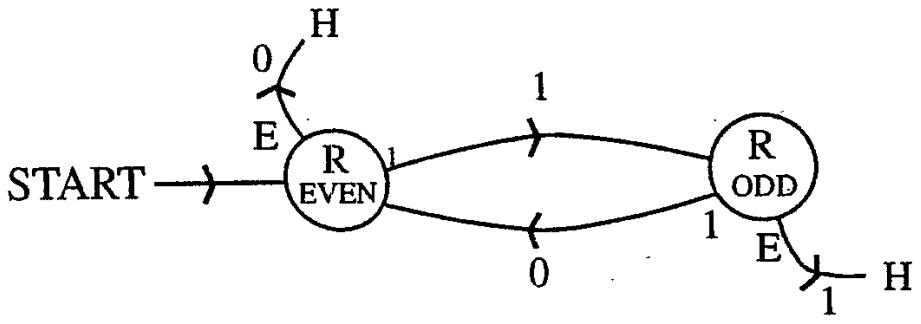


Fig. 3.11 A Turing Machiue Parity Counter

This is essentially the same as Figure 3.5, the FSM which does the same job. Where the FSM has a stimulus, the TM has the contents of a cell. In these diagrams, both are written at the point of contact of lines and circles. Where the FSM spits out a response, which we wrote on the arrow linking states, the TM overwrites the cell contents, what it writes being noted on the arrow. The state labels of both FSMs and TMs are written inside the circles. The major differences are that, firstly, we have to know where the machine starts, which we do by adding an external arrow as shown; and we have to show when it stops, which we do by attaching another arrow to each state to allow for the machine reading *E*, each arrow terminating in a "Halt". More subtly, we also have to describe the direction of its motion after each operation. It turns out that machines whose direction of motion depends *only* on their internal state — and not on the symbols they read — are not fundamentally less capable of carrying out computations than more general machines which allow the tape symbols to influence the direction of motion. I will thus restrict myself to machines where motion to the right or left depends solely on the internal state. This enables me to solve the diagrammatic problem with ease: just write *L* or *R*, as appropriate, inside the state box. In this case, both states are associated with movement to the right.

I have gone on at some length about the rather dumb parity machine as it is important that you familiarize yourself with the basic mechanics and notation of Turing machines. Let me now look at a more interesting problem, that of building a parenthesis checker. This will illustrate the superiority of Turing machines over finite state machines. Suppose we provide our Turing machine with a tape, in each cell of which is written a parenthesis (Fig. 3.12):

....E(())()(()))(()E....

Fig. 3.12 Input Tape to the Parenthesis Checker

Each end of the string is marked with a symbol *E*. This is obviously the simplest way of representing the string. How do we get the machine to check its validity? One way is as follows. I will describe things in words first, and come back to discuss states and diagrams and so forth in a moment. The machine starts at the far left end of the string. It runs through all the left brackets until it comes to a right bracket. It then overwrites this right bracket with an *X* — or any other symbol you choose — and then moves one square to

the left. It is now on a left bracket. It overwrites this with an X, too. It has now canceled a pair of brackets. The key property of the X's is that the machine doesn't care about them; they are invisible. After having canceled a pair in this way, the machine moves right again, passing through any X's and left brackets, until it hits a right bracket. It then does its stuff with the X again. As you can see, in this way the machine systematically cancels pairs of brackets. Sooner or later, the head of the machine will hit an E — it could be either one — and then comes the moment of truth. When this happens, the machine has to check whether the tape between the two Es contains only X's, or some uncanceled brackets too. If the former, the string is valid, and the machine prints (say) a 1 somewhere to tell us this; if the latter, the machine prints 0, telling us the string is invalid. Of course, after printing, the machine is told to halt.

If you think about it, this very simple procedure will check out any parenthesis string, irrespective of size. The functioning of this machine is encapsulated by the state diagram of Figure 3.13 (following Minsky [1967]):

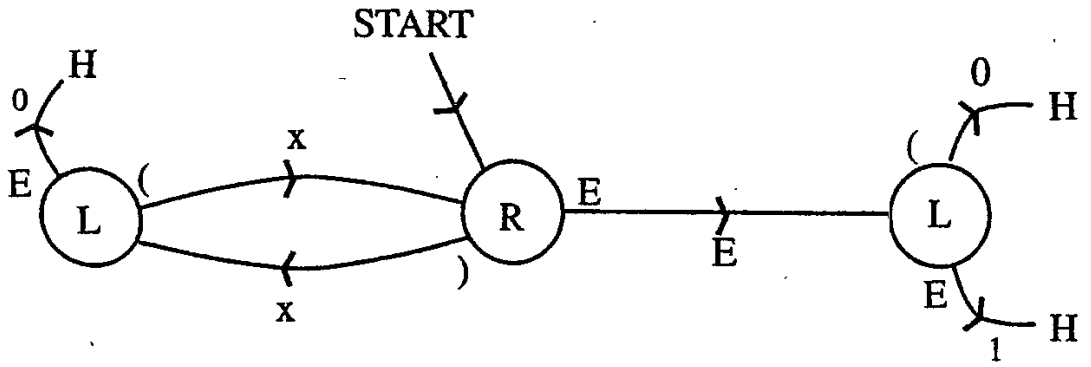


Fig. 3.13 The Parenthesis Checker State Diagram

Note how the diagram differs from that for an FSM: we have to include start and stop instructions, and also direction of motion indicators. In fact, this machine, unlike the parity counter, requires two different left-moving states.

Now that you have some grasp of the basic ideas, you might like to try and design a few Turing machines for yourself. Here are some example problems to get you thinking.

Problem 3.5: Design a unary multiplier. "Unary" numbers are numbers written in base 1, and are even more primitive than binary. In this base, we have only

the digit 1, and a number N is written as a string of N 1's: 1 = 1, 2(base 10) = 11(base 1), 3 = 111, 4 = 1111, and so on. I would like you to design a Turing machine to multiply together any two unary numbers. Start with the input string:

00...E 1111....1 B 1111 ....11 E...00
m n

which codes the numbers being multiplied, m and n and separates the two numbers with the symbol B. The goal is to end up with a tape that gives you mn. It might look something like this:

...00 E 0000...0 B XXX...X E YYYY...Y 00...
m n mn

where Y is some symbol distinct from 0, 1, X, E and B. You can consider the given tape structure a strong hint as one way in which you could solve the problem!

Problem 3.6: We have discussed binary adders before. I would now like you to design a Turing machine to add two binary numbers, but only for the case where they have the same number of bits (this makes it easier). You can start with the initial tape:

...00 A 1101..1 B 1001..0 C 000...
m n

for numbers m and n with the field of the two numbers delineated by the symbols A, B and C. I will leave it to you to decide where the machine starts, how it proceeds, what its final output looks like, where it appears, and so on.

Problem 3.7: If you're finding these problems too easy, here's one that is much harder: design a Turing machine for a binary multiplier!

Problem 3.8: This last problem is neat: design a unary to binary converter. That is, if you feed the machine a string of 1's representing a unary number, it gives you that number converted to binary. The secret to this problem lies in the mathematics of divisors and remainders. Consider what we mean when we talk of the binary form of an n-bit number $N = N_n N_{n-1} \ldots N_1 N_0$. By definition we have:

$$N = N_n.2^n + N_{n-1}.2^{n-1} + \ldots + N_1.2 + N_0$$

We start with N written in unary — i.e. a string of N 1's — and we want to find the coefficients N_i, the digits in binary. The rightmost digit, N_0, can be found by dividing N by two, and noting the remainder, since:

$$N = 2.X + N_0$$

with X easily ascertained. To find N_1, we get rid of N_0, and use the fact that:

$$X = 2.Y + N_1$$

That is, we divide X by two and note the remainder — N_1. We just keep doing this, shrinking the number down by dividing by two and noting the remainder, until we have the binary result. Note that, since N is an n-bit number, by definition N_n *must be 1.*

If we are given the number N in unary form, we can simulate the above procedure by grouping the 1's off pairwise and looking at what is left. Let us take a concrete example. Use the number nine in base ten, or 111111111 in unary. Pair up the 1's:

(11) (11) (11) (11) 1

Clearly, this is just like dividing by two. There is an isolated digit on the right. This tells us that N_0 is 1. To find N_1, we scratch the righthand 1 and pair up the pairs in the remaining string:

(11 11) (11 11).

This time, there is no remainder: N_1 is 0. Similarly, we find that N_2 is 0. We have now paired up all our pairs and pairs of pairs, and the only thing left to do is tag a 1, for N_3, to the left of the number, giving us 111111111 (unary) = 1001 (binary).

I will leave it up to you to implement this algorithm with a Turing machine. You have to get the thing to pair off digits, mark them as pairs and check the remainder; and then come back to the beginning and mark off pairs of pairs, and so on. Marking pairs is probably best done by starting at the left end of the string and going to the right, striking out every other digit and replacing it with an *X* symbol. When the machine gets to go through the string again, it ignores the *X*'s and strikes out every other 1 again. This method, suitably refined, will work! I leave it to you to figure out the details. Don't forget that you have to get the machine to start, perform the conversion, write its output and then stop.

3.5: More on Turing Machines

I would now like to take a look at a fairly complicated Turing machine that bears on a different aspect of computing. Earlier in these lectures I pointed out that computers were more paper pushers than calculators, and it would be nice to see if we can build a Turing machine that performs filing, rather than arithmetic, functions. The most primitive such function is looking up information in a file, and that is what we are going to examine next. We want a machine that first locates a file in a file system, then reads its contents, and finally relays these contents to us[4].

We will employ the following Turing "filing system", or tape (Fig. 3.14), which we are to feed into our machine:

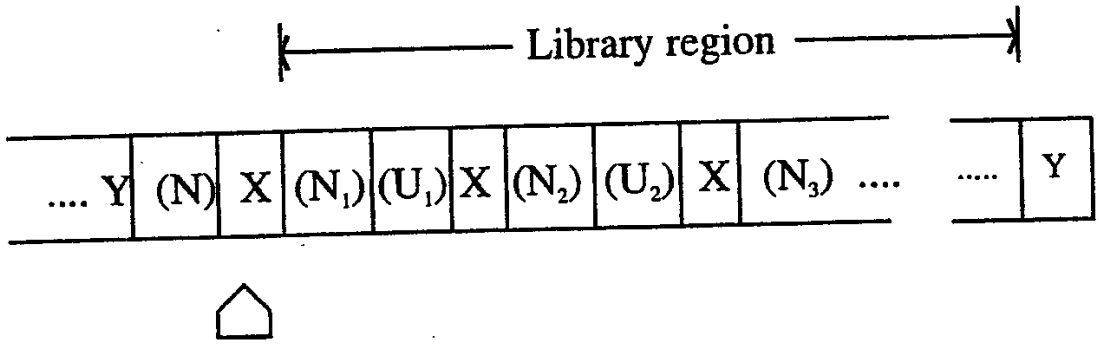


Fig. 3.14 Input Tape to the Locating Machine

[4]Our discussion closely follows Minsky [1967]. [RPF]

This is a bit schematic. The *X*-symbols this time play the role of segregating various file entries; there is one entry between each pair of *X*'s. Each entry comprises a name (or address), "*N*", and contents, "*U*", both of which we take to be binary strings, one digit per tape square as usual. We have attached to the left hand end of the tape the name of a file which we want the machine to read for us, and denoted the left end of the tape by a symbol *Y*. To the left of this is a string of zeroes; the same is true at the right-hand end of the tape. The machine is to start where marked to the right of the name *N* of the file we want to find.

The first task confronting the machine is that of locating the right file. It does this by systematically comparing each file name in the list with the target name, working from left to right, until it finds the correct one. How should it do this? For ease of understanding, suppose we have the following filing tape (Fig. 3.15):

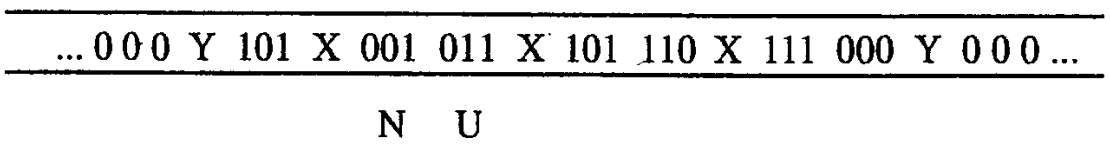


Fig. 3.15 A Sample Filing Tape

For convenience, we are taking both the name strings and the data strings to be of the same length, three bits. We want to read the contents of file (101) which we'll call the target file. Now it might seem that the best thing to do is the following: assign to each possible target a distinct state of the Turing machine. This will give us at most eight states. The machine starts in the state 101 dictated by the target file name and goes to the first file from the left, and looks at the name. If there is a match, all well and good. If not, it goes to the next file on the right, checks that, and so on. In this way, the machine smoothly moves from left to right until it hits the correct address. However, the problem with such a machine is that it has only eight states and will only be any good for three-bit filing systems: it has no universality of application. We want a single machine that can handle any size of filename. To achieve this, the machine must compare each filename with the target on a sequential, digit-by-digit basis, laboriously shuttling between the two until a mismatched digit is found, in which case it goes onto the next filename, or until a complete match is found, when (say) it returns to its starting point. To keep track of those parts of the

tape it has already considered, the machine would do the now-familiar trick of overwriting digits with symbols which it subsequently ignores, just as we did with the parenthesis checker. By assigning different symbols to 0's and 1's — *A*'s and *B*'s, say — we can keep track of which were 0's and 1's; if we wanted to come along tomorrow and use the file again, we could, only we would find it written in a different alphabet. We could then reconstruct the entire original file by overwriting the new symbols with 0's and 1's.

Minsky's solution for a locating Turing machine is shown in Figure 3.16:

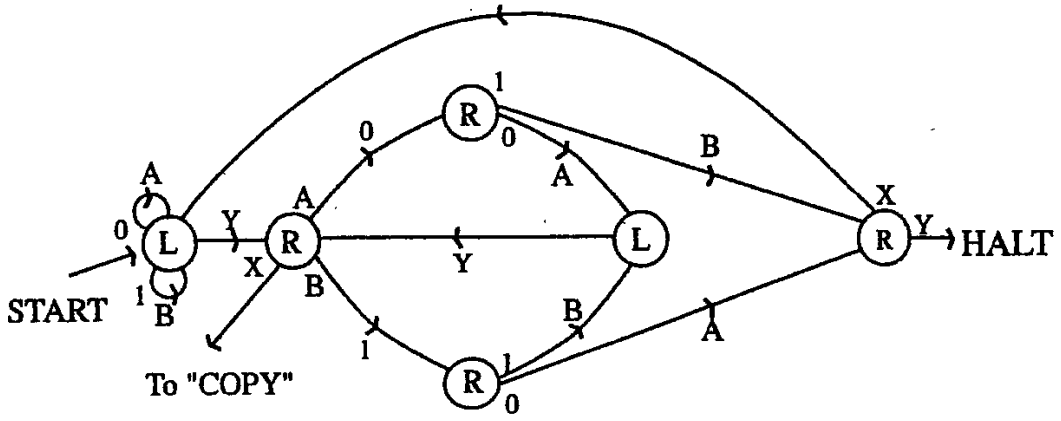


Fig. 3.16 The Locating Machine

There is a loose end in this diagram, pointing to "copy". This represents the stage at which the machine has located the correct filename and is wondering what to do next. We will shortly show how we are going to get it to copy the information in the file to a point of our choosing on the tape.

For the moment let us stick with our location machine and look in more detail at how it works. The head starts on the first *X* to the right of the target number. As the loop instructions show, the machine then heads left, changing the 0's and 1's in the target to *A*'s and *B*'s respectively. This may seem a little bizarre, but there is a point to it, as we will see. Eventually, the machine hits the *Y*. It then goes into a new state, and as is clear from the diagram, it will start moving right. It will first encounter one of the *A*'s or *B*'s it has just written: it overwrites this with the original digit (this *definitely* seems bizarre, but it will

make sense!), a 0 or 1, and moves right again. It now enters one of two states in which it will only recognise a 0 or 1: not an *A* or *B*. If it hits an *A* or *B*, it will ignore it, keeping on moving right – in other words, it is going to pass right through the remainder of the rewritten target string, having in a sense "noted" the first digit of the string. This is why we overwrote the 0's and 1's of the string with *A*'s and *B*'s. It will also pass straight through the *X* it encounters and go on to the first filename to be checked.

Now comes the crucial sequence of operations. The machine is going to hit either a 1 or a 0, and how it reacts depends on how it has been primed – i.e. on the state it is in as a result of reading the first target digit. There are two possibilities. Firstly, if the digit it hits is different from the first target digit, so the filenames do not match from the outset, the machine overwrites the digit as appropriate, and then moves to the right until it hits the next *X*, denoting the end of the file. It then starts to move to the left, overwriting the contents of the rest of the file with *A*'s and *B*'s. It passes through the leftmost *X*, zips through the target filename (*A*'s and *B*'s are invisible to it), changes the first digit to an *A* or *B*, and hits *Y*. This is a cue for the whole process to start again: only now it goes to the next filename. Sooner or later, the first target digit and that of the checked filename will match, and this is the second possibility we must consider.

When a match occurs, the machine overwrites the matching digit, and enters a state in which it moves back left until it encounters the *Y*. Then, it goes forwards, overwrites the second target symbol with the correct digit, and then moves on to the files. It checks the second digit for a match, and so it goes on. Working through the machine diagram, you should be able to convince yourself that the tape above would ultimately be converted into the tape of Figure 3.17:

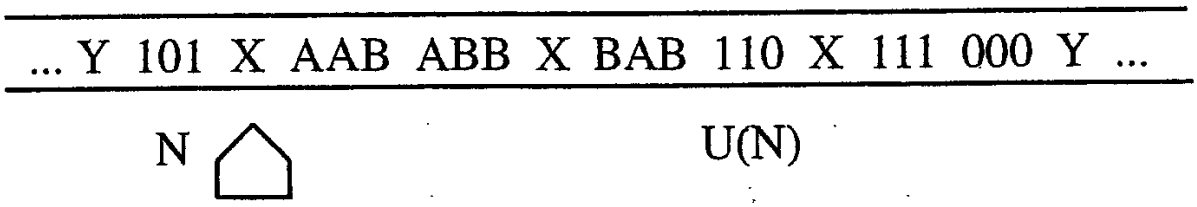


Fig. 3.17 Output Tape from the Locating Machine

Note that the head has returned to its starting point, and the effect of its activities has been to change all 0's and 1's between the start and the end of the

desired filename (but not the contents of the file) to *A*'s and *B*'s. (There is the important possibility that the target filename cannot be found, because we have typed it in wrongly, say, and in this case the machine head will end up on the *Y* at the far right; as the diagram indicates, at this juncture it is instructed to "Halt".)

As I have said, there is a "loose wire" on our diagram, representing a feed to a copy machine: we have our file, now we want to know what to do with it! True to the spirit of Turing machines, we are going to copy it slowly and laboriously to another part of the tape. That is, *you* are: the copying machine is shown in Figure 3.18, and its input tape is the output tape of the location machine. Have fun figuring out how it works!

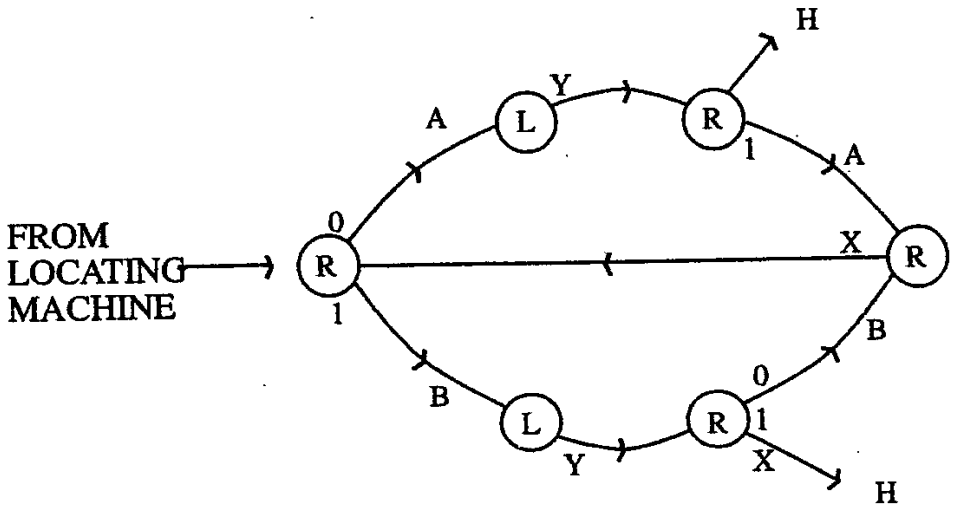


Fig. 3.18 The Copy Machine

A cute feature of this machine is that it copies the contents of the file into the block containing the target filename on the original tape; that is, the target string is overwritten. (We can do this because we chose to have filenames and contents the same size.) The end result of this machine operating on its tape is the tape of Figure 3.19:

... Y BBA X AAB ABB X BAB BBA X B11 000 Y ...

U

Fig. 3.19 Output Tape from the Copy Machine

I will finish this section by giving you a couple more Turing problems.

Problem 3.9: Make a Turing machine which starts with a blank tape and ends up with all the binary numbers written on it in succession, separated by "*Y*'s", with the restriction that after you write the terminating *Y*, you never change the number again. An additional restriction you might impose is that the machine does not even look at terminated numbers.

Problem 3.10: Design a machine which recognizes only, and all, sequences of the form

101100111000111100000.....1"0".

That the machine has "accepted" such a tape is indicated by its halting and leaving the tape blank after its machinations. More generally, we define an arbitrary sequence as "acceptable" by a Turing machine if the machine eventually halts with a blank tape. We can extend this notion to cover finite state machines. Design a Turing machine that accepts exactly the set of sequences accepted by any FSM. (Hint: use the FSM functions *F* and *G* to make Turing quintuples.)

3.6: Universal Turing Machines and the Halting Problem

Let us return to the reason why we are studying Turing machines. I said earlier that if you had an effective procedure for doing some computation, then that was equivalent to it being possible in principle to find a Turing machine to do the same computation. It is useful to talk in terms of functions. Suppose we start with a variable x, and we take a function of that variable, $F(x)$. We say that $F(x)$ is *Turing computable* if we can find a Turing machine $\mathbf{T}_F$ which, if fed a tape on which x is written, in some representation — binary, unary, whatever — will eventually halt with $F(x)$ printed on the tape. Every other effective procedure that anyone else has been able to cook up has turned out to be equivalent to this — the general recursive functions are Turing computable, and vice versa — so we can take "Turing computable" to be an effective synonym for "computable".

Now it may be the case that for some values of x, the Turing machine might not halt. This is weird behavior, but it might happen. Many functions — such as x^2 — are called "complete", meaning that for all values of x we plug into our machine, it will halt with the value of the function written on the tape. Functions for which this is not true are called "partial". In such cases, we have

to alter our operational definition of the function as follows: if, for a value x, the machine stops, we define the value of the function to be $F(X)$; if the machine does not stop, we *define* the value of the function to be *zero*. This does *not* mean that if we put x into F we get zero, in the way that putting $x = 3$ in the function $(x\text{-}3)$ gives us zero. Here, "zero" is just a useful label we attach to $F(x)$ when our Turing machine does not quit its computing. This redefined function is complete in the sense that we can assign *some* numerical value to it for *any* x.

A question naturally arises: can we say, in advance, which values of x might cause our machine to hang up? In some cases, the answer is yes. For example, there may be times when the machine goes into a recognizable infinite loop, perhaps shuttling between a couple of states and not achieving anything, and we can then say for sure that it will never stop. But in general, we cannot say in advance when a particular value of x is going to give us trouble! *Put another way, it is not possible to construct a computable function which predicts whether or not the machine T_F halts with input x.* In seeing why this is so, we shall appreciate the power of Mr. Turing's little machines.

I have flagged what is to follow in the penultimate sentence of the previous paragraph. I have raised the question of whether there is a computable function which will tell us whether or not $\mathbf{T_F}$ halts for input x. But, if there is such a function, by definition it must be describable by a Turing machine. This concept, of Turing machines telling us about other Turing machines, is central to the topic of *Universal* Turing machines to which we now turn.

We can pose the question we have set ourselves in the following way. Suppose we have a machine which we call **D**. As input, **D** takes a tape which contains information about $\mathbf{T_F}$ and $\mathbf{T_F}$'s initial tape (that is, information about X). Machine **D** is required to tell us whether $\mathbf{T_F}$ will halt or not: yes or no. Importantly, **D** must always write the answer and halt, itself. What we now do is introduce another machine **Z**, which reacts to the output from **D** in the following way:

> *If T_F halts (**D** says "yes"), then **Z** does not.*
> *If T_F does not halt (**D** says "no"), then **Z** does.*

We then get **Z** to operate on itself and find a contradiction! Let us expand on this argument.

To begin our quest for **D**, we first need to look at how we get one Turing machine to understand the workings of another. We need to characterize a given machine **T**, and its tape *t*; there are several ways of doing this. We will choose a description in terms of quintuples (Table 3.3):

Initial		Final		
State	Read	State	Write	Move
Q	S	Q′	S′	d (= L or R)

Table 3.3 Quintuple Description of a Turing Machine

We want to build a universal machine that is capable of imitating any **T**. In other words if we feed it information about **T** and about **T**'s tape *t*, our universal machine spits out the result of **T** acting on *t*. We will characterize our universal machine — call it **U** — in terms of quintuples in similar fashion to T. Let these quintuples for **U** be written $(q,s;\ q',\ s',\ d')$ and note that they must suffice for all possible machines **T** that we want **U** to imitate: q,s, etc. must not depend on the specifics of **T**. A constraint we shall impose on our machines is that the tape symbols S, S', s, s' must be binary numbers. An arbitrary Turing machine **T** will come with an arbitrary set of possible symbols, but with thought you should be able to see that we can always label the distinct symbols by binary numbers and work with these (e.g. if we had 8 symbols, each could be redescribed by a three-bit binary string)[5].

The basic behavior of **U** is simple enough to describe. (Our discussion again closely follows Minsky [1967].) We need **U** to imitate **T** step by step, keeping a record of the state of **T**'s tape at each stage. It must note the state of **T** at each point, and by examining its simulated **T**-tape it can inform itself what **T** would read at any given stage. By looking at the description it has of **T**, **U** can find out what **T** is supposed to do next. Minsky nicely relates this process to what *you* would do when using a quintuple list and a tape to figure out what a Turing machine does. The universal Turing machine **U** is just a slower version of you!

[5]In fact, as an exercise, examine how you would reprogram a Turing machine **T** that operated with 2^n symbols to become a machine **T′** operating on 0 and 1. Hint: where **T** had to read one

Let us supply **U** with the tape shown in Figure 3.20:

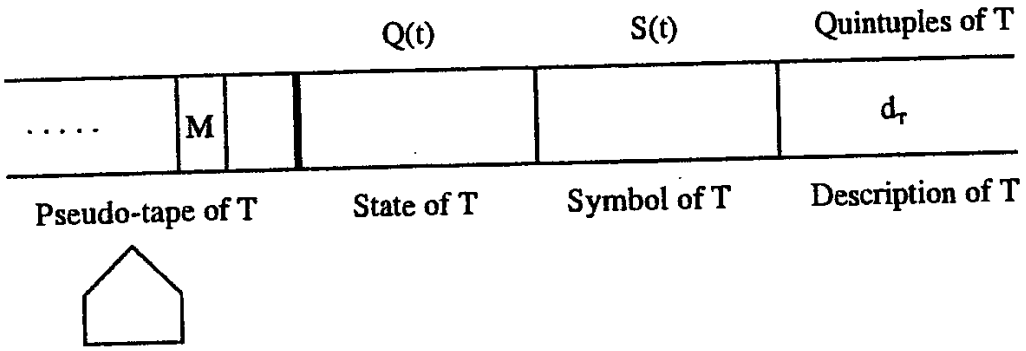


Fig. 3.20 Input Tape to the Universal Turing Machine

The infinite "pseudo-tape" on the left is **U**'s working space, where **U** keeps track of what **T**'s tape looks like at each stage of its simulation. Choosing to have it infinite only to the left is not essential, but simplifies things. The marker *M* tells **U** where the tape head of **T** currently is on *t*. To the right of this working space is a segment of tape containing the state of **T**; then, next right is a segment containing the symbol just read by **T**; and finally, to the right again, is a region containing the description of **T**. This description of **T**, which we denote as d_T, comprises a sequential listing of the quintuples of **T**, written as a binary sequence (Fig. 3.21):

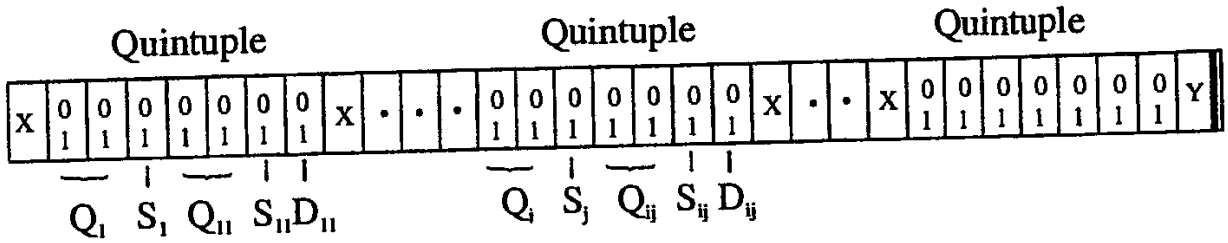


Fig. 3.21 The Description d_T of T for U's Tape

Each quintuple is segregated from the next by the symbol *X*. To start **U** off, we need to tell it **T**'s initial state Q_0 and the symbol S_0 it reads first. Let us assume that **U**'s tape head is initially over the leftmost *X* as shown in Figure 3.22:

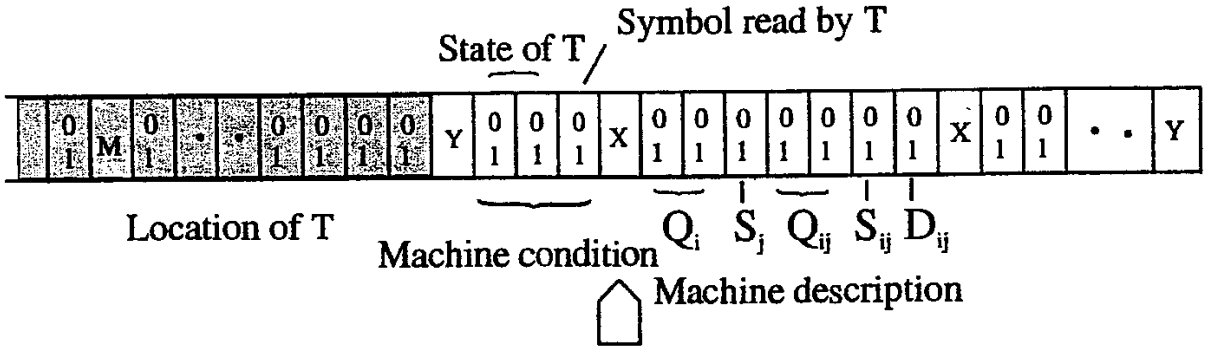


Fig. 3.22 Starting Position for the Tapehead of U

Essential to the operation of **U** are the locating and copying operations we described earlier. In general terms **U** operates as follows. First, **U** looks in the section of the tape describing the action of **T** for a given Q and S, exactly as we did with the locating machine: the set (Q,S) can be regarded as the filename of the file containing the relevant quintuple. As with the locator, on its way this operation changes all the 0's and 1's it encounters to A's and B's. After it finds the relevant pair (and changes them to A's and B's), it returns to the leftmost X.

The next stage involves the copy machine. **U** moves to the right until it hits the first set of 0's and 1's; because of the way we have set up the tape, these represent the three remaining parts of the quintuple specified by Q and S. These are the new state of **T**, the symbol it writes (on the pseudo-tape in position M) and its subsequent direction of motion. The machine then copies A's and B's representing both the new Q and new S into the machine condition region in the middle of the tape. It remembers the direction of motion d (L or R, represented as A or B). The machine now heads left until it reaches M. Once there, it erases M and temporarily overwrites it with the direction d (A or B). It then moves right, changing all A's and B's to 0's and 1's on the way (leaving an A or B in M's old location). Finally, it moves to the immediate left of the leftmost X, erases the symbol S that is there (but remembers it) and prints the special symbol V in its place (this is all that V is used for).

The machine now enters its final phase. It shifts left until it encounters the A or B that we stored in M; this represents the direction d in which **T** should next move. The machine overwrites the A or B with the S it has remembered, and then moves left or right depending on the instruction d. It reads the symbol of the square it is now on, remembers it and prints an M in its place. It then shifts right until it reaches the V, which it replaces with the remembered symbol. Now the sequence starts all over again.

What the machine has done is simulate one cycle of T's operation: it has started off in a certain state Q and a given input symbol S; it has then changed state, written a new symbol and moved on to the next symbol dictated by T. U continues like this until it has mimicked T completely. Importantly, U has a halt state: it recognizes when T has halted, and proceeds to stop itself.

The description of U given above, due to Minsky, requires U to have 8 symbols and 23 states. So that you can appreciate the beauty of his machine, we reproduce it in full in Figure 3.23. You should not find it too hard to break it down into its constituent sub-machines.

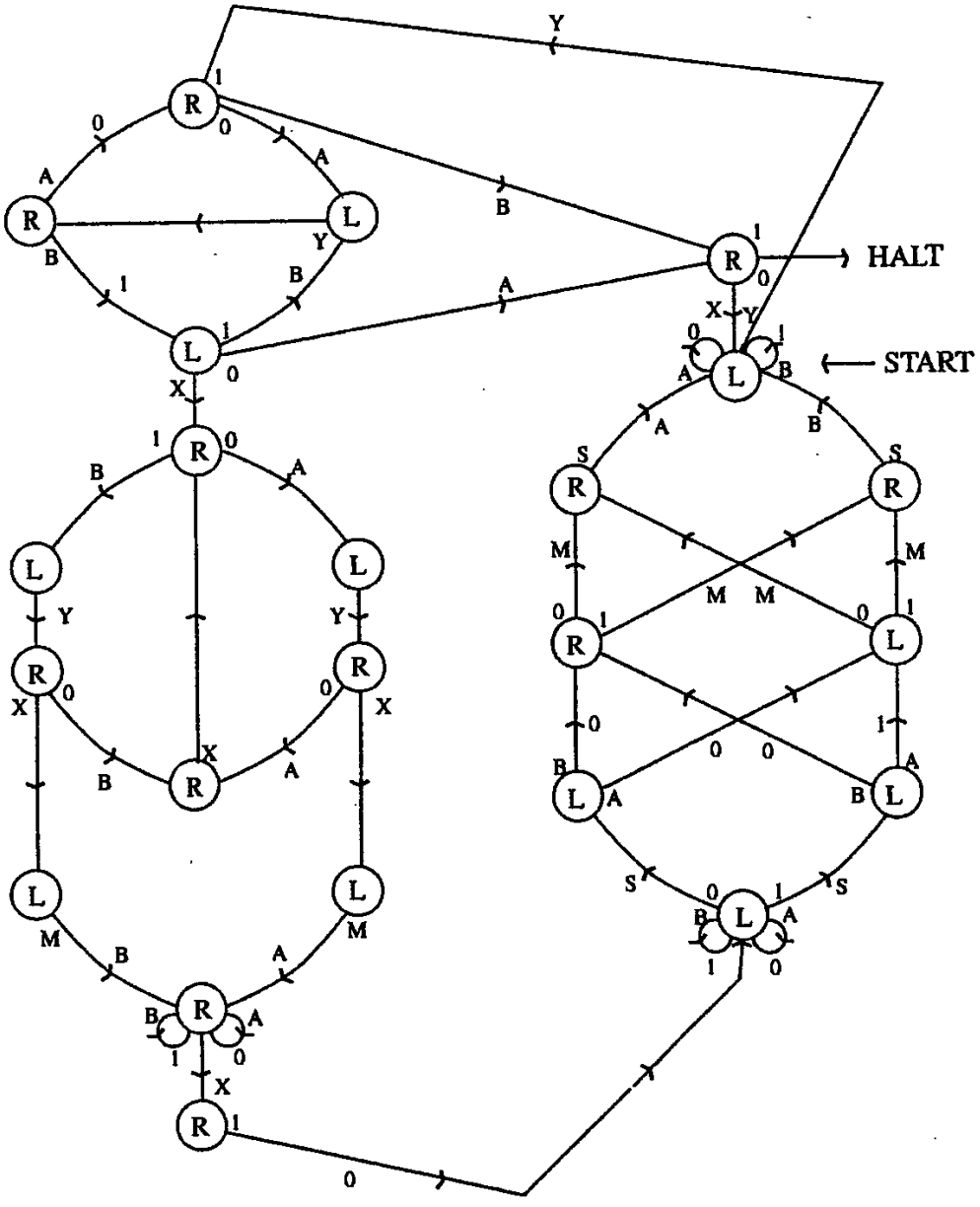


Fig. 3.23 A Universal Turing Machine

It is possible to build a UTM with the same number of symbols but just 6 states. If one wants to get tricky, there are ways of using the same state for more than one purpose, to minimize the number of states required. A UTM can be built with just two states and lots of symbols, or two symbols and lots of states. It is surprising that such a general purpose machine should require so few parts for its description; surely a machine that can do *everything* should be enormously complicated? The surprising answer is that it's not! How efficient one can make a UTM is an entertaining question, but has no deep significance.

Let us now turn to the real reason why we have been interested in demonstrating the existence of a UTM. We have asked whether it is possible to build a machine that will tell us whether a Turing machine **T** with tape *t* will halt, for all **T** and *t*. We can clearly rephrase this as a halting problem for a universal machine **U**. Let us define a new machine **D**, which is just **U** with the added property that it tells us whether or not **T** halts with tape *t*, and that it can do this for all machines **T** and all tapes *t* (Fig. 3.24):

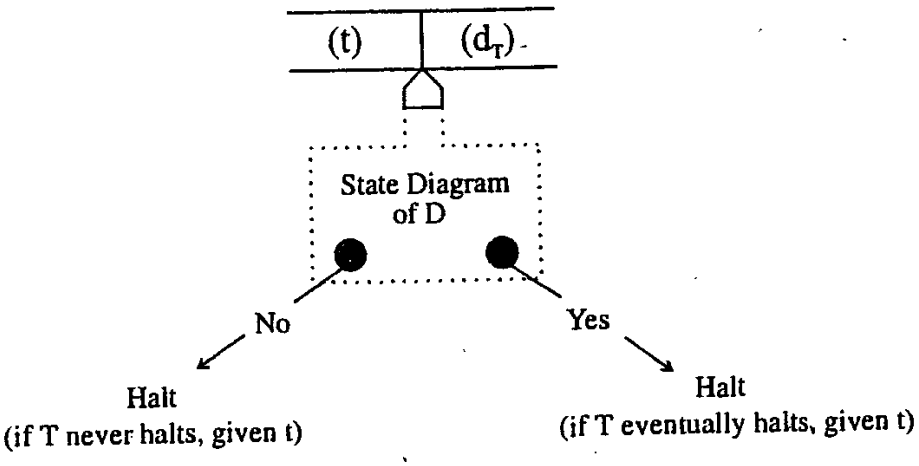


Fig. 3.24 Universal Machine D with tape t and d_T

In other words, **D** *always* halts with an answer. Can such a machine exist? The answer is no! We can actually show that **D** is an impossible dream, and we do this by picking a machine **T** and a tape *t*, for which **D** cannot do what it is supposed to.

Information about **T** and *t* are fed into a universal machine in the form d_T, the quintuple description of **T**, and the information on the tape *t* (see Fig. 3.24). Now for no apparent reason, let us see what happens if we let the tape *t* contain the description d_T. We now enhance our machine **D** slightly and introduce another machine **E**. This new machine only requires as input a tape containing d_T; it then copies d_T onto a blank part of the tape and now behaves like machine

D with an input tape containing $t = d_T$ and d_T. **E** will now behave the same way as **D**, and halt giving the answer "yes" if **T** halts when reading its own description: otherwise, **E** will answer "no" (Fig. 3.25). Whatever the case, **E** always halts.

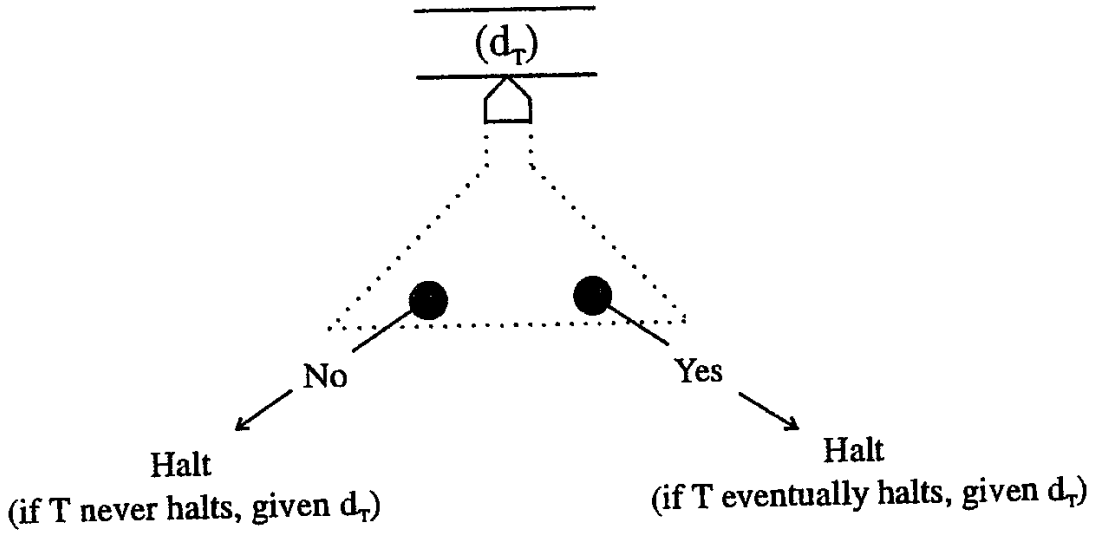


Fig. 3.25 Universal Machine E with input tape d_T

Now we introduce a modified version of **E** which we shall call **Z**. Our new machine **Z** has two extra states that are used to prevent **Z** from halting if **E** takes the "yes" route (Fig. 3.26):

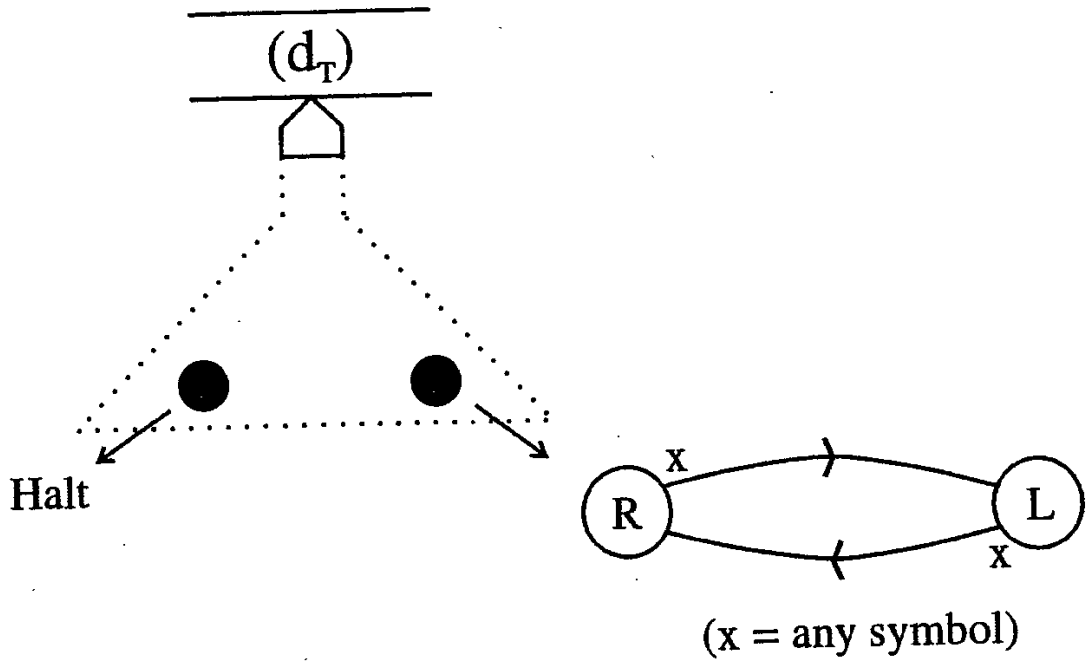


Fig. 3.26 Universal Machine Z

Thus **Z** has the property that, if **E** spits out the answer "yes", it does not halt; whereas if **E** spits out "no", it also gives us a "no" and does halt (i.e. **Z** = **E** in this case). So, **Z** halts when we feed it d_T, if **T** applied to d_T does *not* halt, but does not halt if **T** applied to d_T *does*. Now comes the crucial step. Let us write a description d_Z for **Z**, and substitute **Z** for **T** in the foregoing argument. We then deduce that:

Z applied to d_Z halts if and only if **Z** applied to d_Z does not halt.

This is a clear contradiction! Going back through our argument, we find that it is our assumption that **D** exists that is wrong. *So there are some computational problems (e.g. determining whether a UTM will halt) that cannot be solved by any Turing machine.* This is Turing's main result.

3.7 Computability

There must be many uncomputable functions. How many are there? We can gain some insight into this by considering a counting argument. Consider computable real numbers: by which we mean those whose binary expansions can be printed on a tape, whether the machine halts or not. We can show that there are many more real numbers than computable real numbers since the latter are countable, while the former are not. We call a set "countable" if we can put its elements in one-to-one correspondence with elements of the set of positive integers; that is, if we can label each set member by a unique integer. Two examples of countable sets are the even and rational numbers:

Even numbers	0	2	4	6	8	10....				
	0	1	2	3	4	5				
Rational numbers	(1/2)	(1/3	2/3)	(1/4	2/4	3/4)				
	1		2	3		4	5	6		

The real numbers, however, are not countable. We can supply a neat proof of this as follows. Let us suppose the opposite. Then we would be able to pair off the reals with the integers in some way, say as follows:

Integer	Real
1	0.124
2	0.015
3	0.53692
4	0.8003444
5	0.334105011
6	0.3425.....

The exact assignment of real numbers to integers, and we have chosen a weird one here, is arbitrary; as long as we have one real number per integer, and all the reals are accounted for, we are OK. However, this cannot be so! To see why, we will find a real number that cannot be on our list. In the above list, I have underlined certain digits: the first digit of the first number, the second digit of the second, the third of the third, and so on. We define a new number using these: all we require is that the nth digit of this number *differs* from the nth digit in our list. The real number:

0.22741....

going on forever, is just such a number. We have obtained this by adding one to each of the underlined digits. (We can include the rule "9+1=0" to make this a consistent procedure or we can use other procedures entirely to generate new real numbers.) What have we achieved? By construction, the above number differs from the mth number in our correspondence list in its *m*th digit, and this is true for all m — that is, for all integers. Hence, we have found a real number that cannot appear on our list. So by "diagonalization" as it is called (referring to the "diagonal" line we can draw through all of the underlined numbers above) we have shown that the real numbers are not countable.

Turing machines, however, *are* countable. To see this, consider the tape description d_T of a machine **T**. We can consider this to be a string of binary symbols unique to the machine if we ignore the spacings between quintuple listings. The resulting binary number serves to uniquely label the machine by an element of the set of integers. On the other hand, if we define a function $f(n)$ to be 1 if the nth Turing machine halts and 0 otherwise, then clearly this function is not computable, as we have seen from the Halting Problem. There are many other examples.

Let us return to the subject of effective procedures and make a few comments. Although we have tended to portray effective procedures as

algorithms that enable us to calculate things, in reality many such procedures are of little practical use — they might require too much tape for their execution, for example, or some other extravagant use of resources. A procedure might take the age of the Universe to complete yet still be technically "effective". In practice we want procedures that are not just effective but also efficient. The word "efficient", of course, is not easy to define precisely and so we end up leaving the clean and unambiguous world of logic and entering that of the real world of the comparatively dirty and vague — or exciting and interesting — depending on your viewpoint! Many problems in "artificial intelligence", such as face recognition, involve effective procedures that are not efficient — and in some cases, they are not even very effective!

Sometimes we do not strictly need effective procedures at all. It might be the case, for example, that you can ask a question and, while I cannot give you a sure answer, I can answer it with a probability of correctness of $(1 - 10^{-20})$. You might be quite happy with such good odds. There is nothing particularly bad about uncertainty. An obvious, and rather uninteresting, example of this would be if you asked me whether a given number x was divisible by some other number y. I could simply say "no", and if y is big enough, the odds are in my favor that I am right: to be precise, the odds are 1 in y that a randomly chosen number is divisible by y. The principle here is that you can know a lot more than you can prove! Unfortunately, it is also possible to think you know a lot more than you actually know. Hence the frequent need for proof.

A related, but more interesting problem, is the question of whether or not a given number n is prime. An effective procedure for this might involve taking all prime numbers up to $n^{1/2}$ and seeing if any divide n; if not, n is prime. This is fine, and rather neat, for small n, but when we get to the big numbers it becomes impractical. A better test is a probabilistic one. This uses one of Fermat's famous theorems:

$$a^p = a \bmod p \tag{3.4}$$

What this means is that, for any number a and prime p, if we divide a^p by p, we get the remainder a. So for example, we write:

$$3^5 = 243 = (48 \times 5) + 3$$

The idea behind the method is to take a large value of a, and calculate a mod

p. For large p, the odds are good that p is not a prime and that this quantity does not equal a since there are so many possible remainders. (The actual odds are not simple to calculate, but you get the idea.) However, if p is huge — something of the order of 10^{200}, say — how do we calculate a^p? Well, we don't actually need this number: we only need the remainder after division by p. Why this is so I will leave as an exercise for you! (Don't worry about the general case: do it for a nine-digit p.)

Another similar problem deals with factorization: I give you a number m, and tell you that it is the product of two primes, $m=pq$. You have to find p and q. This problem does not have an effective procedure as yet, and in fact forms the basis of a coding system. It is possible to build our ignorance of the general solution of this mathematical problem into ciphering a message. The moment some clever guy cracks it — and people have gotten up to 72 digit m's so far — the code is useless, and we'd better find another one.

Before leaving the subject of computability, I want to make some remarks about the related topic of "grammars". In mathematics, as in linguistics, a grammar is basically a set of rules for combining the elements of a language, only the language is a mathematical one (such as arithmetic or algebra). It is possible to misapply these rules. Consider the following statements:

$$(a + b)\ c \qquad\qquad a + b(\ c$$

Within the context of arithmetic, only the first of these makes sense. The second, however, does not: the parenthesis is wrongly, even meaninglessly, placed. An interesting general question in computing is whether we can build machines that will test mathematical (and other) expressions for their grammatical correctness. We have seen one example: the parenthesis checker. This checked a very simple grammar involving) and (and the only grammatical rule was that strings of parentheses balanced. But remember it took a Turing machine to do this: a finite state machine was not up to it. Now there are certain classes of grammar that FSMs can check — for example, strings of ones, 1111111.... where valid strings have to have even numbers of digits, for example — but the abilities of this type of machine are limited. We can actually draw up a table relating types of grammars to the machines required for their analysis (Table 3.4):

Language	**Description**	**Example**	**Machine required**
Finite enumerable	A list of acceptable expressions	*ab*, *abc*	Memory (table look-up)
Regular language	Regular expressions built with *, ∨, ∧, ( )	ab^*c, *=any no. of repetitions, incl. none. $a(b \vee d)^*c$	Finite state machines (a theorem)
Context free	Language generated by production rules which admit recursion	$a^n b^n$ (not $a^n b^m$ where $n \neq m$)	An in-between machine: a push-down automaton. Has one "stack" inside — a pile of paper with a spring underneath, can only take off the top one
General recursive	Computable functions	$a^n b^m c^q$	Turing

Table 3.4 General Grammars and Their Machine Implementation

It is sad that Turing machines are so easy to make that we have to leap over all this pretty theory. Nevertheless, in the design of compilers (which involve the interpretation of languages) the use of such theory is so fundamental that you might find further study of it worthwhile.

We will finish our look at computability with an interesting problem discovered by Post as a graduate student in 1921. Consider a binary string, say 10010. It is arbitrary. Given the string, play with it according to the following rules: read the first three digits; if the first is 0, delete all three and add 00 to the end of the string; if the first is 1, delete all three and add 1101 to the end. So with our string we would have

```
10010
--- 101101
  ---   1011101

        -----
```

The question is this: does this process go on forever, stop, or go on periodically? The last I heard, all tested sequences had either stopped or gone into a loop, but that this should be so generally had not been proved. It is an interesting issue because it has been shown that a so-called "Post machine" — one which takes a string g and writes a result $h(g)$ depending on the first digit g_i of the string — can act as a Universal machine and do anything a Turing machine can do!