

9. EXTENDED TURING MACHINES

§ 9.1. The Basic Turing Machine and Its Extensions

Following Alan Turing we will adopt the Turing machine as the model for computability. We shall define something to be computable if and only if it can be computed by a Turing machine. The implication will be that anything that can be done on the most powerful computer we can construct, using any programming language that we could possibly dream up, can be done on a Turing machine.

It would be nice if we could *prove* this, but of course that would require us to first define what can be done on any computer using any programming language. Since this would have to include, not only existing computers and existing programming languages, but also computers and languages that could be constructed in the future.

The best that we can do is to use the Turing machine as the definition of computability and then to provide sufficient evidence to show how powerful a Turing machine can be.

The basic Turing machine, as described in the previous chapter, seems extremely primitive. With just one track, one head and two characters programming even simple tasks seem extremely clumsy and require great ingenuity. If only we had multiple tracks, multiple heads and a larger set of characters we would have a much more powerful machine.

In this chapter we consider these possible extensions. But we will then show that whatever can be done on these extended Turing machines can also be done on the basic one, though much more clumsily, requiring many more states and many more steps. Certainly having many tracks, many heads and many characters would be much more efficient, but efficiency is not relevant here. It is not as if we are going to build Turing machines to carry out real computing tasks. Our concern is simply whether or not something can be done at all. We want to know whether there are computing tasks that are logically impossible.

Another reason for including this chapter applies perhaps more to the mathematics student. In mathematics we learn not only to solve problems, but also to prove that certain problems can or cannot be solved. And frequently we do this by showing that a general class of problems can be solved by reducing them to a narrower class. This is a fundamental paradigm in mathematics — going from the general to the more restricted.

In this chapter we shall use our imagination to extend the basic Turing machine, giving it all sorts of extra bells and whistles which at first sight might seem to make it more powerful. But by showing that these more powerful machines can be reduced to the basic type we will be showing that they are no more powerful in terms of what they can do.

The **Basic Turing Machine** is the one we have been discussing. It has one head, one track, two characters (including blank) and two directions (left and right). But its easy to think of extensions which might give us a more powerful machine.

We could extend the number of characters. We could add extra tracks with the possibility of going up and down as well as left or right. We could even have multiple heads. Certainly

these extensions would make programming a Turing Machine more convenient. But, as we shall show, they do not make a more powerful machine. Anything that can be done on one of these extended machines can be simulated on the Standard Turing Machine.

One, very simple extension would be to allow a “neutral direction” N where the head stays on the current square. This would be convenient in situations when we want to halt on a certain character. One we reach it we have no way, on the basic machine, of just staying there. We are forced to move left and then right (or right and then left). It would be nice to have a third possibility for the movement of the head – N = no movement.

In such a Turing Machine each instruction would have the form cDs where c is a character, s is a state and D is either L, R or N. The instruction 1N5 would mean that the machine prints a “1”, stays on the same square, and goes to state 5.

More convenient, perhaps, but not more powerful. Quite clearly any Turing Machine with such “neutral direction” instructions could easily be converted to one without.

§ 9.2. Multiple Tracks

Suppose we had two tracks instead of one. As well as being able to go left or right we could also go up and down. There is the question of what happens if the head is on the top track and is told to go up, or on the bottom track and is instructed to go down. We would have to stipulate that the machine halts in such cases, just as when it is told to go to a non-existent state.

We would thus have 4 directions L, R, U and D (and N too if we like, but we’ll leave that out for the moment).

Example 1:

The following machine implements the function SWAP(m, n) for positive natural numbers m, n. The two numbers are written in unary notation with their left-most 1’s in the same column. The head starts and finishes on the left-most 1 on the bottom tape.

	0	1	
0	0U1	1U2	read bottom track
1	0D3	0D5	0 read on bottom
2	1D5	1D6	1 read on bottom
3	0L3	1L4	halt if both tracks
4	0R7	1L4	read 0
5	1R0	0R0	swap 1 with 0
6		1R0	“swap” 1 with 1

We can implement any 2 track machine on a single track one by numbering the squares on a single track by the integers, positive and negative. We can then regard the even numbered squares as the bottom track and the odd numbered ones as the top track.

	...	-5	-3	-1	1	3	5	7	9	...	
	...	-6	-4	-2	0	2	4	6	8	...	

corresponds to

...	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	...
-----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	-----

The following table shows the corresponding movements on the 2 track and single track machines.

2 track	1 track
UP	LEFT
DOWN	RIGHT
LEFT	LEFT twice
RIGHT	RIGHT twice

An instruction of the form cUs on the 2 track machine becomes cLs and cDs becomes cRs. An instruction of the form cLs or cRs requires the addition of extra states so that we can move left or right twice.

For example suppose we have the following row in the table for our two track machine.

7	0U9	1R2
---	-----	-----

The instruction 0U9 would be changed to 0L9 on the basic version. To implement 1R2 we would need an extra state s. We would print the "1" and move right in the first step, going to a new state 7'. Then we would ignore what was on this square and move right again.

7	0L9	1R7'
7'	0R2	1R2

If we had the following row in the table for the two track machine we would need to introduce two extra states.

7	0L9	1R2
---	-----	-----

The corresponding Turing machine would contain the following states:

7	0L7'	1R7''
7'	0L9	1L9
7''	0R2	1R2

Example 2: Convert the extended Turing machine in example 1 to a single track one:

	0	1
0	0L1	1L2
1	0R3	0R5
2	1R5	1R6
3	0L3	1L4
3'	0L3	1L3
3''	0L4	1L4
4	0R7	1L4
4'	0L7	1L7
4''	0L4	1L4
5	1R0	0R0
5'	0R0	1R0
5''	0R0	1R0
6		1R0
6'	0R0	1R0

This illustrates how each left or right instruction on the two track machine gives rise to an extra state when we convert to a single track machine. However we can clearly combine states 3'' and 4'' and states 5', 5'' and 6' can also be combined, resulting in an 11 state machine. (We would, of course, renumber the states 0 to 10.)

If we had an extended Turing machine with any finite number of tracks we could use a similar method to convert it to an equivalent single track machine. We would treat the columns as successive block of k squares. A DOWN instruction would become a RIGHT instruction and an UP instruction would become a LEFT instruction. A LEFT instruction would be simulated by a sequence of k LEFT's on the single track machine, and a RIGHT instruction would become a sequence of k RIGHT's. Each LEFT or RIGHT instruction on the k -track machine would give rise to k additional states, though many of these might be able to be amalgamated.

More tracks means more convenience but no more functionality. And since Turing machines exist purely as a model for the computing process, convenience is not an issue. Whatever can be done on a multi-track machine can be done on the basic version.

It is even possible to convert a Turing machine with infinitely many tracks into a single track version. We would have to use a more sophisticated way of making squares on the two-way infinite machine correspond with those on the single track version. We could not take blocks of squares corresponding to the columns like we did when we have a finite number of tracks. Instead we could start at one of the squares and number the others by moving around in some sort of spiral. Working out which square to move to next would be quite tricky, but it can be done.

§ 9.3. Multiple Heads

Two heads are better than one. Suppose we have a single track Turing machine, but with two read/write heads. Only one can be active at any given time so we shall suppose that we associate a particular head with each state. When we move to a given state our table will tell us which head is active.

Example 3: The following Turing machine with one track and two heads will swap any pair of positive integers. The integers are to be in unary, separated by a 0 with head 0 on the leftmost 1 of the first block and head 1 on the leftmost 1 of the second. So, for example, the pair (2, 4) will be represented by:

	0	1	1	0	1	1	1	1	0	
		↑			↑					
		0			1					
	head	0	1							
0	0	1L1	1R0	Plug up the separating blank						
1	0	0R2	1L1							
2	1	0L4	1R3	Alternately move the heads						
3	0		1R2							
4	0		0L5	Make a new separating blank						
5	0	0R6	1L5							
6	1	0R7	1L6	Reset the heads						

How can we simulate a multi-head Turing machine by a basic one. The difficulty is that the basic Turing Machine has only one read/write head, which can't be in two places at once. To keep track of the two heads we need to mark their positions on the tape. But a mark in the middle of data might become confusing. So we'll use extra tracks, one for each head. On each of these tracks we'll put a single "1" in the column where the corresponding head is supposed to be on the data track.

With two heads we'll need three tracks, one for the data and one for each of the heads. For example the single track tape in example 3 would appear as follows on the single head machine.

pointer for head 0	0	0	1	0	0	0	0	0	0	0
data track	0	0	1	1	0	1	1	1	1	0
pointer for head 1	0	0	0	0	0	1	0	0	0	0

We can implement a two head, single track, Turing machine by a one head three track machine. We do this by inserting a copy of FIND before each row in the table, where we go to the track that corresponds to the newly active head, find the "1" pointer and then go back to the data tape.

This conversion will greatly increase the number of states and this number will be even greater when the 3 track machine is converted to the single track basic Turing machine. But, as you will remember, this doesn't matter. The question is only "can it be done" and the answer is "yes, it can".

§ 9.4. Multiple Characters

The two characters 0 and 1 ought to allow us to represent integers in binary but there would be no way we could decide where the binary string starts and finishes. If only we had a couple of extra characters.

If we had three characters, say 0, 1 and #, the tables for our Turing machines would have three columns. With the extra character we can now represent positive integers in binary.

The **binary notation** for a natural number encloses the binary string that represents that number between a pair of #'s. The read/write head starting and finishing on the first character. So, for example, the string 1011 would be written on the tape as #.1011#. Remember that the first character of the binary representation of a positive integer is "1" and that the binary representation of the number zero is the string "0". So zero would appear on the tape as #.0#.

Example 5: Construct a Turing Machine to add 1 to a binary number.

Solution:

	0	1	#	
0	0R0	1R1	#L1	move to right-hand end
1	1L2	0L1	1L3	carry
2	0L2	1L2	#R4	no carry
3	#R4			extend string where necessary

How can we implement a multi-character Turing machine using only two characters. The simplest way to do this is to code the characters in binary and write these in the columns of a multi-track machine. Then we can convert the multi-track machine to a single track one.

Suppose we had a single track Turing Machine with four characters 0, 1, # and *. We could implement this on a Standard Turing Machine, using just 0's and 1's by using two tracks. Each column would represent a single character, expressed in binary. For example we could code 0 as 00, 1 as 01, * as 10 and # as 11.

...	#	1	0	*	1	1	0	1	#	...
-----	---	---	---	---	---	---	---	---	---	-----

would become:

...	1	0	0	1	0	0	0	0	1	...
...	1	1	0	0	1	1	0	1	1	...

We would start on the top row and then move down. This will determine which character is being read and therefore which instruction is to be carried out. The printing part of the instruction would require going back up to the top track.

Example 6:

Suppose we have the following row in the table for a four-character Turing machine:

	0	1	*	#

4	1L5	#L2	#R9	*L3

This state would be expanded 5 states for the 2 track 2 character equivalent.

	0	1

4	0D4'	1D4''
4'	1U4'''	1U4''''
4''	1U4'''	0U4''''
4'''	0L5	1R9
4''''	1L2	1L3

A Turing Machine with 3 characters would be very useful for processing binary strings. As with natural numbers in binary form we would enclose the string between two #'s and have the head start on the square immediately to the right of the left #. Normally this would be the first character of the string, but for the null string it would be the second # since the null string would appear as #.# on the tape.

Example 7: Construct a Turing Machine, using standard Binary Notation, to reverse a binary string.

Solution: We create a mirror image of the original string and then erase the original string. We do this by temporarily replacing a characters by a # while we go to the mirror image and deposit it in the correct place and then return to where it came from and reinstate it. There is no need to reinstate the characters of the original string but we do it because a slight modification will produce a TM that makes a second copy of a binary string. Until the character has been reinstated we need to remember what it was. We do this by having parallel sets of states, one (states 3 to 7) if it was a 0 and a second lot (here states 8 to 12) if it was a 1. State 13 then erases the original string.

	0	1	#		
0	0R0	1R0	#R1	Move to end of string	
1	#L1		#L2	Create an extra #	
2	#R3	#R8	0R13	Mark & remember character	
3	0R3	1R3	#R4	IF 0 go to right # and change to 0 and create new right marker; reinststate the 0 in the original string.	
4	0R4	1R4	0R5		
5	#L6				
6	0L6	1L6	#L7		
7	0L7	1L7	0L2		
8	0R8	1R8	#R9		IF 1 go to right # and change to 1 and create new right marker; reinststate the 1 in the original string.
9	0R9	1R9	1R10		
10	#L11				
11	0L11	1L11	#L12		
12	0L12	1L12	1L2		
13	0R13	0R13	#R14	Erase the first copy	

§ 9.5. A Universal Turing Machine

The Turing machine is a model for the computing process with which we can prove facts about computability. This means that anything which can be computed on a real computer, no matter how complex or how advanced, can be done by a Turing machine. Of course real computers have more elaborate hardware than the Turing machine's paper tape. It's hard to see how one could display graphics on a Turing machine. But the processing that lies behind computer graphics can certainly be achieved on Turing's primitive little machine.

Can we prove this? Not really. The problem is the difficulty of defining what is meant by a "real computer". Instead we take as a definition of computability to mean something that can be computed by a Turing machine.

Is this a good definition? Over the years many great minds have tried to formulate a definition of computability and several very different definitions have been proposed. The fact that each of them has been proved to be equivalent to every other definition makes us believe that we have the "right" definition.

But because it is difficult to program even fairly simple tasks on a basic Turing machine it is easy to believe that they *are* more limited than a computer that runs a more sophisticated language. To show that Turing machines are capable of quite complex tasks we shall show how a Turing machine can be constructed which can simulate any given Turing machine — a universal Turing machine.

Such a universal Turing machine would be supplied with the instructions of a Turing machine, *M*, together with some data, *D*. The universal machine would then simulate the operation of *M* on this data and produce the same output.

Such a machine would come closer to the sort of computers we are familiar with. While the usual Turing is described as a single-purpose device with a single program "hard-wired" into its instruction table, the universal machine is a multi-purpose device that is capable of running any Turing machine in the same way that a typical computer takes a program as part of its data.

There may seem to be something vaguely incestuous about a program being able to simulate any program written in the same language, why it could even simulate itself, simulating itself And is it really possible for a program, big as it might be, to simulate even bigger programs of the same type?

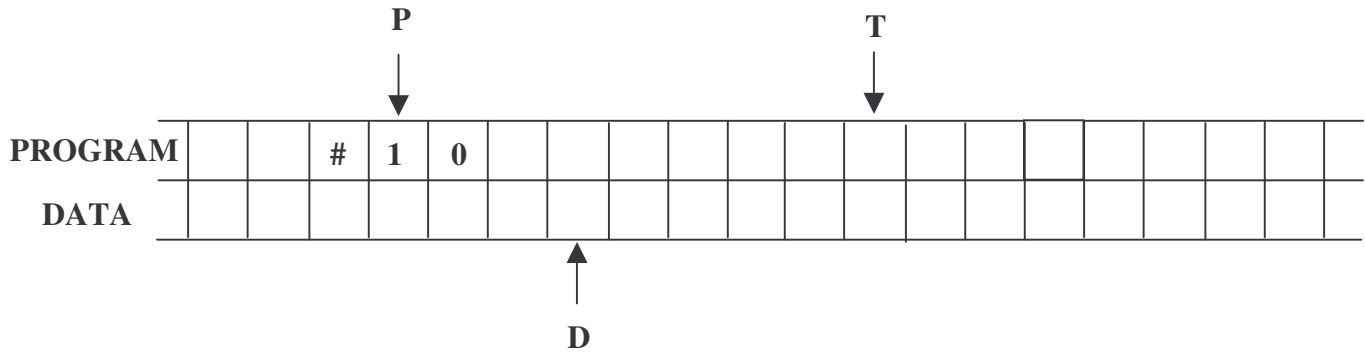
Yet this is not the logical contradiction that it might seem. In fact this has become standard practice in the practical world of computing. The compilers for the powerful programming language C++ are usually written in C++ itself!

So to support the claim that whatever can be achieved computationally by any real programming language can be achieved by a Turing machine we will outline the construction of a universal Turing machine. If that doesn't convince you perhaps you should try to construct a C++ compiler as a Turing machine! Even that could be done if one had the patience.

We shall construct it as a 2-track, 3-head, 3-direction, 3-character machine which, as we know, could be converted to a basic Turing machine.

The three directions are L (left) R (right) and N (no movement). The three characters are 0, 1 and #. The upper track is called the Program Track and will contain the instruction table for an arbitrary Turing machine *M*. The lower track, called the Data Track will simulate the tape for *M*.

The three heads are the D-head, the P-head and the C-head. The D-head will operate only on the data track and will simulate the head for *M*. The P-head will operate only on the program track and will act as a Program Pointer by pointing to the first character of the first instruction for the current state. The C-head, also operates only on the program track and is used as a counter.



We now have to specify how the Turing machine programs/tables are laid out on the Program Track. We code L as 0 and R as 1 so the first two characters of each instruction represent the character to be printed and the direction. Then we write the state in unary, as a sequence of 1's. Separating each instruction we have a 0. The instructions are represented in pairs, giving the two instructions for each state – the one to be followed if the head reads 0, followed by the instruction to be obeyed if the head is reading a 1. The entire program is enclosed between a pair of #'s.

Suppose we have the 2-state TM:

	0	1
0	1L1	1R0
1	0R2	1L1

We write these instructions as binary strings as follows:

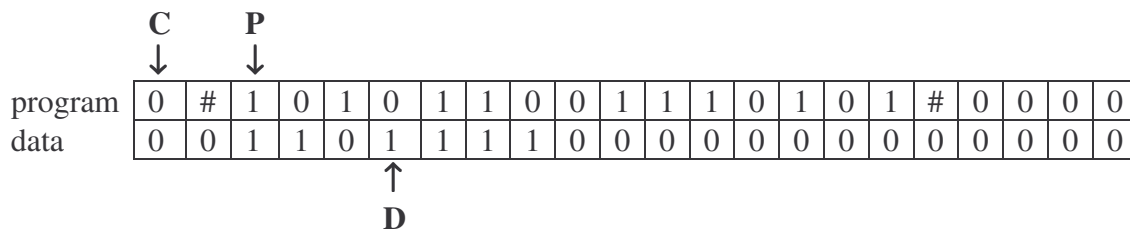
	0	1
0	101	11
1	0111	101

The first bit is the 0 or 1 that is to be printed. The second bit represents the direction, with 0 = L and 1 = R. The follows a string of 1's to represent the state, with n 1's representing state n.

We now combine these into a single list with instructions separated by 0's and surround the entire string between a pair of #'s: #101011001110101#

When the Universal Turing machine begins the heads are placed as follows:

D-head	On the square on the Data Track that the head of the machine M is scanning
P-head	Immediately to the right of the left # on the Program Track
C-head	Immediately to the left of the left # on the Program Track



The operation of the Universal Turing Machine operates in a loop that processes one instruction. First the D-head is activated and reads the character on the data track. But it stays on that square because we have yet to print something there.

We now have to locate the next instruction to be obeyed. If a 0 was read on the data track the P-head will already be pointing to the start of that instruction. But if a 1 was read we have to move to the right by one instruction (equivalent to going to the second column).

Moving one instruction to the right consists of moving to the right twice along the Program Track (past the character to be printed and the direction) and then continuing to move right (through the block of 1's that represents the next state) until a 0 is read. This will be the separator between the first and second instructions for that state. The P-head then moves right one more square.

So the Universal Turing Machine, up to the point where it accesses the correct instruction, is:

	head	0	1	#	
0	D	0N4	1N1		Read the character on the Data Track.
1	P	0R2	1R2		If a 1 was read, move right ...
2	P	0R3	1R3		... move right again and then ...
3	P	0R4	1R3		... move right till a 0 is read and move right one more square.

The P-head will now be reading the character to be printed. The Universal Turing Machine now has to remember whether this is a 0 or a 1 and then it has to communicate this information to the D-head which does the actual printing. The remembering is done by having two separate states, one state if a 0 is to be printed and another if it is to be a 1. After this character is printed the D-head doesn't move because the Universal TM has to go back to the program to find out which way to move.

So the Universal Turing Machine continues as follows:

	head	0	1	#	
4	P	0R5	1R6		Read the character to be printed.
5	D	0N7	0N7		Print a 0 ...
6	D	1N7	1N7		... or a 1.

Now the Universal TM reads the code that represents the direction the D-head has to move: 0 meaning "left" and 1 meaning "right". So the Universal TM continues as follows:

	head	0	1	#	
7	P	0R8	1R9		Read the direction.
8	D	0L10	1L10		Move the D-head left ...
9	D	0R10	1R10		... or right.

At this point the Universal TM is ready to read the number for the new state. Here is where the C-head comes in. The state number must be copied to the left of the left # on the Program Track.

	head	0	1	#	
10	P	0R12	1R11		Copy the state to the left of the left #.
11	D	1L10			

To go to the new state we have to move the Program Pointer to the beginning of the first instruction for that state. But since we must count from the very beginning the P-head has to be reset back to state 0.

	head	0	1	#	
12	P	0L12	1L12	#R13	Reset the Program Pointer.

Now, as the C-head moves through each 1 the P-head moves one state (i.e. two instructions).

	head	0	1	#	
13	C	0R13	0R14	#L0	Decrement the counter.
14	P	0R15	1R15		Move past the first instruction.
15	P	0R16	1R16		
16	P	0R17	1R16		
17	P	0R18	1R18		Move past the second instruction.
18	P	0R19	1R19		
19	P	0R13	1R19	#L20	

When the C-head encounters a # in state 13 this means that the Program Pointer has moved to the required state and so the whole process begins again in state 0. But when the P-head reads a # in state 19 this means that we have been sent to a non-existent state. In other words, the machine we are simulating is halting. So the Universal Turing Machine needs to halt.

Putting this altogether we have a Universal Turing Machine with only 20 states. Of course once we converted this to a basic Turing machine, with only two characters, one track, one head and no N direction, the number of states would probably be many hundreds. But such a Universal Turing Machine does indeed exist.

EXERCISES FOR CHAPTER 9

Ex 9A1: Convert the following TM to a standard one.

	0	1
0	1L1	1N0
1	0N1	1L2

Ex 9A2: Convert the following 2-track TM to a single one.

	0	1
0	1L1	1U0
1	0R1	1D2

Ex 9A3: Construct a Turing Machine using Standard Binary Notation, to copy a binary string. (So, for example, #1011# would become #1011#1011# with the head on the first character after the initial #.)

Ex 9A4: Construct a Turing Machine with 3 characters 0, 1 and # which locates a “1” under the following conditions. There is only one # on the tape and somewhere to the right of it is a “1”. The rest of the tape is blank. The head starts at, or to the left of, the #. When the TM halts the tape is unchanged and the head stops on the “1”. (It carries out the same function as the TM FIND, but because of the special circumstances it is a much simpler machine.)

Ex 9A5: Construct a Turing Machine on the alphabet {0, 1, #}, where 0 denotes a blank, which takes a non null string of 1’s and #’s and transfers the right-most symbol to the left-hand end. Thus ... 0 0 0 1 # # # 1 # 0 0 0 ... becomes ... 0 0 0 # 1 # # # 1 0 0 0 ...
The head is located on the left-most (non-blank) symbol at the beginning and the end of the process.

Ex 9A6: Design an extended TM with one track, one head and three characters 0, 1, #, to compute the following functions. Input and output are to be in binary as follows. The binary string that represents n is enclosed between # characters and the head is placed immediately to the right of the left #. So, for example, 0 is represented by #.# and 6 is represented by #.110#.

- (i) $f(n) = n + 1$;
- (ii) $g(n) = 2n$.

Ex 9A7: Design an extended TM with two tracks, one head and two characters 0, 1, to compute the function $f(n) = 2n + 1$ for $n \geq 2$. Input and output are to be in binary on the lower track with the head on the leftmost 1. The upper track has two 1’s, placed directly above the first and last character of the binary string on the lower track.

Ex 9A8: Design an extended TM with one track, one head and three characters 0, 1, #, to compute the functions $\text{HALVE}(n) = \text{INT}(n/2)$. Input and output are to be in binary as follows. The binary string that represents n is enclosed between # characters and the head is placed immediately to the right of the left #. So, for example, 0 is represented by #.# and 6 is represented by #.110#.

WARNING: Make sure that it works correctly for $n = 0$ and $n = 1$.

SOLUTIONS FOR CHAPTER 9

Ex 9A1:

	0	1
0	0L1	1L0 _R
0 _R	0R0	1R0
1	0L1 _R	1L2
1 _R	0R1	1R1

which, after renumbering, becomes:

	0	1
0	0L2	1L1
1	0R0	1R0
2	0L3	1L4
3	0R2	1R2

Ex 9A2:

	0	1
0	1L1 _L	1L ₀
1 _L	0L1	1L1
1	0R1 _R	1R2
1 _R	0R1	1R1

which, after renumbering, becomes:

	0	1
0	1L1	1L0
1	0L2	1L2
2	0R3	1R4
3	0R2	1R2

Ex 9A3:

	0	1	#	
0	0R0	1R0	#R1	Put in extra #
1	#L2			
2	0L2	1L2	#R3	
3	#R4	#R9	#L14	Read char
4	0R4	1R4	#R5	IF 0 copy and reinstate
5	0R5	1R5	0R6	
6	#L7			
7	0L7	1L7	#L8	
8	0L8	1L8	0R3	IF 1 copy and reinstate
9	0R9	1R9	#R10	
10	0R10	1R10	0R11	
11	#L12			
12	0L12	1L12	#L13	Reset head
13	0L13	1L13	0R3	
14	0L14	1L14	#R15	

Ex 9A4:

	0	1	#	
0	0R0	1R0	#R1	Find #
1	0R1	1R2		Find 1
2	0L3			Reset head

Ex 9A5:

	0	1	#	
0	0L1	1R0	#R0	Go to right-hand end
1	0R4	0L2	0L3	Read character
2	1L4	1L2	#L2	IF 1 move it to left
3	#L4	1L3	#L3	IF # move it to left
4	0R5			Reset head

Ex 9A6:

(i)

	0	1	#
0		1R2	#L1
1	#L3		#R4
2	0R2	1R2	0R1
3	0L3	1L3	#R4

(ii)

	0	1	#
0	0R0	1R0	0R1
1	#L2		0R4
2	0L2	1L2	#R3
3	#L1	1L3	#R4

Ex 9A7:

	0	1
0		1U1
1	0R1	0R2
2	1D3	
3	1U3	1L4
4	0L4	1D5

NB states 0, 2 could be combined to give a 4 state machine.

Ex 9A8:

	0	1	#
0	0R0	1R0	0L1
1	#L2	#L2	#L3
2	0L2	1L2	#R4
3	#R4		

