9. EXTENDED TURING MACHINES

§ 9.1. The Standard Turing Machine and Its Extensions

We define the **Standard Turing Machine**, abbreviated to **STM**, to be the Turing Machine we developed in the last chapter – one tape, one head, two characters). Following Alan Turing we'll adopt the STM as the model for computability. In other words we define something to be **computable** if and only if it can be computed by an STM. 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 an STM.

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. This would have to include, not only existing computers and existing programming languages, but also computers and languages that might be constructed in the future. The best that we can do is to use the Standard Turing machine as the definition of computability and then provide sufficient evidence to show how powerful such a machine can be.

The Standard Turing machine, as described in the previous chapter, seems extremely primitive. With just a single track tape, one read/write head and two characters (mark and blank, or 0 and 1), programming even simple tasks seems extremely clumsy and requires great ingenuity. If only we had multiple tracks on our tape, multiple heads and a larger set of characters we'd have a much more powerful machine – or so we might think.

In this chapter we'll consider some of these possible extensions. But we'll then show that whatever can be done on one of these extended Turing machines can also be done on the STM, 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 isn't relevant here. It's not as if we're 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'll 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'll be showing that they're no more powerful in terms of what they can do.

How might we extend the Standard Turing Machine? We could increase 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 much more convenient. But, as we'll show, they don't make it any more powerful. Anything that can be done on one of these extended machines can be simulated on the STM.

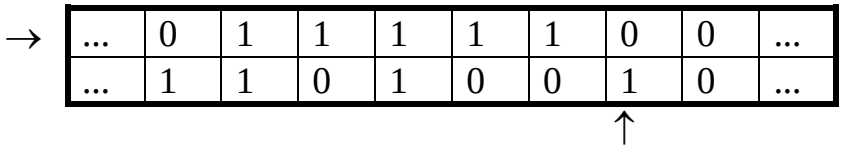
One, very simple extension would be to allow a 'neutral direction' N where the head stays on the current square. This would be convenient in situations where we want to halt on a certain character. One we reach it we have no way, on the STM, of just staying there. We're forced to move left and then right (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 *c*D*s* 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’s the question of what happens if the head is on the top track and it is told to go up, or on the bottom track and is instructed to go down. We’ll stipulate that if the head is on the top track and is told to go UP it moves to the bottom track of the previous column. And, if it is on the bottom track and is told to go DOWN, it moves to the top track in the next column on the right. And we could indicate the position of the head by two arrows, one at the left indicating the row, and one underneath, indicating the column.



Here the head is scanning the ‘1’ that appears in the 1 0 column indicated.

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

***k*-TRACK, N-STATE TURING MACHINE:**
Start in state 0. Halt in state N. *m*D*n* MEANS:
- **PRINT *m***
- **MOVE one square in direction D**

(L = left, R = right, U = up, D = down)
Go UP from top track = go to bottom of previous column.
Go DOWN from bottom track = go to top of next column.
- **GO TO STATE *n***

Example 1:

The following machine implements the function SWAP(*m*, *n*) for positive natural numbers *m*, *n*. The two numbers are written on the two tracks, 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 top tape.

	0	**1**	
0	0D1	1D2	read top track
1	0U3	0U5	0 read on top
2	1U5	1U6	1 read on top
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. We can then regard the even numbered

squares as the bottom track and the odd numbered ones as the top track.

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

corresponds to

...	–5	–4	–3	–2	–1	0	1	2	3	4	5	...

With this arrangement going UP on the 2-track machine converts to going LEFT on the 1-track machine, even if we are on the top track. And going DOWN converts to moving RIGHT on the standard machine. But moving RIGHT on the 2-track machine requires the 1-track equivalent to move RIGHT twice.

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

We add to our STM table the rows:

s′	0Ls	1Ls
s″	0Rs	1Rs

Then we change the instructions in the 2-track table as follows:

$$cUs \rightarrow cLs;$$
$$cDs \rightarrow cRs;$$
$$cLs \rightarrow cLs';$$
$$cRs \rightarrow cRs''$$

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

	0	**1**
0	0D1	1D2
1	0U3	0U5
2	1U5	1U6
3	0L3	1L4
4	0R7	1L4
5	1R0	0R0
6		1R0

Solution: Adding the extra states we get:

	0	**1**	
0	0R1	1R2	Original 2-state machine with $U \to L$ $D \to R$ cLs→cLs′ cRs→cRs″
1	0L3	0L5	
2	1L5	1L6	
3	0L3′	1L4′	
4	0R7″	1L4′	
5	1R0″	0R0″	
6		1R0″	
0′	0L0	1L0	Standard instructions for converting UP into 2 LEFTs.
1′	0L1	1L1	
2′	0L2	1L2	
3′	0L3	1L3	
4′	0L4	1L4	
5′	0L5	1L5	
6′	0L6	1L6	
0″	0R0	1R0	Standard instructions for converting DOWN into 2 RIGHTs.
1″	0R1	1R1	
2″	0R2	1R2	
3″	0R3	1R3	
4″	0R4	1R4	
5″	0R5	1R5	
6″	0R6	1R6	

Clearly many states in the bottom two-thirds are redundant and may be removed:

	0	1
0	0R1	1R2
1	0L3	0L5
2	1L5	1L6
3	0L3′	1L4′
4	0R7′′	1L4′
5	1R0′′	0R0′′
6		1R0′′
3′	0L3	1L3
4′	0L4	1L4
0′′	0R0	1R0

There is no row for 7′′ because it is the HALTING state. So now we can renumber the states 7, 8, 9, giving:

	0	1
0	0R1	1R2
1	0L3	0L5
2	1L5	1L6
3	0L7	1L8
4	0R10	1L8
5	1R9	0R9
6		1R9
7	0L3	1L3
8	0L4	1L4
9	0R0	1R0

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. If we had *k* tracks 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 on the k-track machine 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. This would involve *k* additional states for each state *s* on the *k*-track machine, which we would label as $s^{(1)}$, $s^{(2)}$, …, $s^{(k)}$.

Having 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 standard version.

Perhaps we could have infinitely many tracks! Consider a Turing machine which operates on an infinite plane divided vertically and horizontally into squares. Even this can be converted to an STM but we'd 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 couldn't take blocks of squares corresponding to the columns like we did when we had 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 could be done.

§ 9.3. Multiple Characters

The two characters ‘0’ and ‘1’ ought to allow us to represent integers in binary but there’d be no way we could decide where the binary string starts and finishes. We have to use unary. 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 starts and finishes on the first character. So, for example, the string 1011 would be written on the tape as #**.**1011# (the **.** indicates the position of the head – it scans the symbol immediately to the right).

Remember too 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# .

***k*-CHARACTER, N-STATE TURING MACHINE: USE *k* COLUMNS:**

Example 3: 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 an STM.

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:

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

↑

We'd 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 4:

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 to 5 states for the 2 track 2 character equivalent.

	0	**1**	
	…..	…..	
4	0D4′	1D4″	Digit or symbol?
4′	1U4‴	1U4⁗	0 or 1?
4″	1U4‴	0U4⁗	* or #?
4‴	0L5	1R9	Back on top track – move left or right
4⁗	1L2	1L3	
	…..	…..	

Example 5: 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.

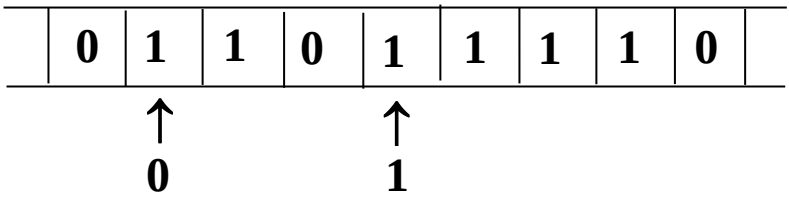
	0	**1**	**#**	
0	0R0	1R0	#R1	Move to end of string
1	#L1		#L2	Create an extra #
2	#R3	#R8	0R13	Mark & remember character
3	0R3	1R3	#R4	IF 0 go to right #
4	0R4	1R4	0R5	and change to 0
5	#L6			and create new right marker;
6	0L6	1L6	#L7	Reinstate the 0
7	0L7	1L7	0L2	in the original string.
8	0R8	1R8	#R9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

§ 9.4. Multiple Heads

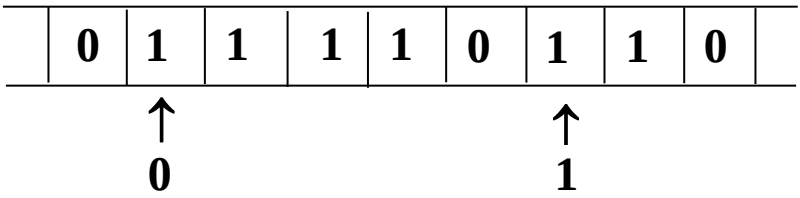
Two heads are better than one. Suppose we have a single track Turing Machine, but with two read/write heads. Our instructions would need to specify which head is active, so we add a column for each state that specifies the active head.

MULTIPLE HEADS **For each state specify the active state.**

Example 6: The following Turing Machine with one track and two heads will swap any pair of positive integers. The integers are to be in unary form, with one head scanning the beginning of each number. So, for example, the pair (2, 4) will be represented by:



and when we've swapped them we'll have:



Note that we don't actually have to move the 1's. We merely relocate the zero. We plug up the separating blank (replace the '0' by a '1' and use head 1 to count the second number, while the first number moves up and reinstates the separating zero in the right place. But, since the position of the heads is part of the output, we have to reset them to their appropriate positions.

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

How can we simulate a multi-head Turing Machine by an STM? The difficulty is that the STM 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 the data might become confusing. So we'll use extra pointer 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 two pointers.

We also add two extra symbols. They will be '#' and '*'. So we'll place the symbol '#' on the pointer tracks to the left of the data. So whenever we need to locate a pointer we move left till we hit the '#' and then right till we reach the '1'. The symbols '#' and '*' will also be used for temporary replacements for '0' and '1' on the data track.

At the start, the head on the 1-track equivalent will be scanning the square on the data track corresponding to the square scanned by the 2-head machine,

Example 7: The single track double head tape in example 3 would appear as follows on the single head multi-track machine:

data track	...	#	0	0	1	1	0	1	1	1	1	0	...
pointer for head 0	...	#	0	0	1	0	0	0	0	0	0	0	...
pointer for head 1	...	#	0	0	0	0	0	1	0	0	0	0	...

We can therefore implement such a 2-head,1-track, 2-character machine by a 1-head, 3-track, 4-character machine. This conversion will greatly increase the number of states and this number will be even greater when the 3 track machine is converted to an STM. But, as you'll remember, this doesn't matter. The question is only "can it be done?" and the answer is "yes, it can!"

We'll create 4 temporary instructions:

- H $\rightarrow$H0, meaning change Head to current position of the Head 0 pointer.
- H $\rightarrow$H1, meaning change Head to current position of the Head 1 pointer.
- H0 $\rightarrow$H, meaning change Head 0 pointer to current position of the Head.
- H1 $\rightarrow$H, meaning change Head 1 pointer to current position of the Head.

Whenever the head changes from 0 to 1 we'll insert the instructions H0$\rightarrow$H and H$\rightarrow$H1.
Whenever the head changes from 1 to 0 we'll insert the instructions H1$\rightarrow$H and H$\rightarrow$H0.

Example 8: Using these temporary instructions we will now convert the 2-head machine in example 3 to a single head equivalent.

2-Head, 1-Track:

	head	**0**	**1**
0	0	1L1	1R0
1	0	0R2	1L1
2	0		1R3
3	1	0L4	1R2
4	0		0L5
5	0	0R6	1L5
6	1	0R7	1L6

1-Head, 3-Track

	0	**1**
0	1L1	1R0
1	0R2	1L1
2		1R3
2′	H0→H	
2″	H→H1	
3	0L4	1R2
3′	H1→H	
3″	H→H0	
4		0L5
5	0R6	1L5
5′	H0→H	
5′	H→H1	
6	0R7	1L6

We can write these temporary instructions in terms of the instructions:
UP, DOWN, PRINT *c*, FIND *c* where *c* is a character.
When finding a character we won't know whether to search left or right. While we can search alternately left and right in ever increasing sweeps, this would result in more complicated code. Instead we search left till we find '#' and then right till we find what we are looking for.

FIND 1 means to search left until we find the '#' and then right till we find '1'.

FIND * means to search left until we find the ‘#’ and then right till we find ‘*’.

H→H0:
DOWN
FIND 1
UP

H→H1:
DOWN
DOWN
FIND 1
UP
UP

H0→H:
DOWN
PRINT *
FIND 1
PRINT 0
FIND *
PRINT 1
UP

H1→H: DOWN
DOWN
PRINT *
FIND 1
PRINT 0

FIND *
PRINT 1
UP
UP

	0	1
0	1L1	1R0
1	0R2	1L1
2		1R3
3	DOWN	
4	PRINT *	
5	FIND 1	
6	PRINT 0	
7	FIND *	
8	PRINT 1	
9	UP	
10	DOWN	
11	DOWN	
12	FIND 1	
13	UP	
14	UP	
15	0L22	1R2
16	DOWN	
17	DOWN	

	0	1
18	FIND 1	
19	UP	
20	FIND 1	
21	UP	
22		0L23
23	0R34	1L23
24	DOWN	
25	DOWN	
26	PRINT *	
27	FIND 1	
28	PRINT 0	
29	FIND *	
30	PRINT 1	
31	FIND 1	
32	UP	
33	UP	
34	0R35	1L34

Finally we must convert these temporary instructions into instructions of the form *c*D*s*. Here *s*′ is the next state after s and s′′ is the next state after that.

Note that when we put our head changing instructions together we can get some simplifications. For example H0→H followed by H→H1 results in an UP followed by a DOWN. Clearly such a pair can be eliminated.

UP

	0	**1**	*****	**#**
s	0Us′	1Us′	*Us′	#Us′

DOWN

	0	**1**	*****	**#**
s	0Ds′	1Ds′	*Ds′	#Ds′

PRINT c

	0	**1**	*****	**#**
s	cNs′	cNs′	cNs′	cNs′

FIND 1

	0	**1**	*****	**#**
s	0L0s	1Ls	*Ls	#Rs′
s′	0Rs′	1Ns′′	*Rs′	#Rs′

FIND *

	0	**1**	*****	**#**
s	0L0s	1Ls	*Ls	#Rs′
s′	0Rs′	1Rs′	*Ns′′	#R′

So our 2-head machine has grown to a 3-track, 4character, single head machine. Many of the cells are inaccessible (especially in the # and * columns) but they have been retained where they have been translated as above, so that you can follow things more easily.

Of course we haven't yet obtained a Standard Turing Machine. We have 4 characters, which will require 2 data tracks, giving 4 tracks altogether if we go binary. Then each L and R will translate into 4 states when we combine these 4 tracks into a single one.

	0	**1**	**#**	*****
0	1L1	1R0		
1	0R2	1L1		
2		1R3		
3	0D4	1D4	#D4	*D4
4	*N5	*N5	*N5	*N5
5	0L5	1L5	#R5′	
5′	0R5′	1N6		
6	0N7	0N7		

7	0L7	1L7	#R7′	*L7
7′	0R7′	1R7′	#R7′	*N8
8	1N9	1N9	1N9	1N9
9	0U10	1U10	#U10	*U10
10	0D11	1D11	#D11	*D11
11	0D12	1D12	#D12	*D12
12	0L12	1L12	#R12′	*L12
12′	0R12′	1N13	#R12′	*R12′
13	0U14	1U14	#U14	*U14
14	0U15	1U15	#U15	*U15
15	0L22	1R2		
16	0D17	1D17	#D17	*D17
17	0D18	1D18	#D18	*D18
18	0L18	1L18	#R18′	*R18
18′	0R18′	1R19	#R18′	*R18′
19	0U20	1U20	#U20	*U20
20	0L20	1R20	#R20′	*R20
20′	0R20′	1N21	#R20′	*R20′
21	0U22	1U22	#U22	*U22
22		0L23		
23	0R34	1L23		
24	0D25	1D25	#D25	*D25
25	0D26	1D26	#D26	*D26
26	*N27	*N27	*N27	*N27
27	0L27	1L27	#R27′	*L27
27′	0R27	1N28	#R27	*R27
28	0N29	0N29	0N29	0N29
29	0L29	1L29	#R29′	*L29

29′	0R29′	1L29′	#L29′	*N30
30	1N31	1N31	1N31	1N31
31	0L31	1L31	#R31′	*L31
31′	0R31′	1N32	#R31′	*R31′
32	0U33	1U33	#U33	*U33
33	0U34	1U34	#U34	*U34
34	0R35	1L34		

This can be simplified by removing instructions that can never be reached. However for those that are translations of temporary instructions they have been retained to make it easier to follow.

§ 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 is 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 define 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 the others makes us believe that we have the 'right' definition.

But because it's difficult to program even fairly simple tasks on a basic Turing Machine it's easy to believe that they *are* more limited than a computer that runs a more sophisticated language. To add evidence to the claim that Turing Machines are capable of quite complex tasks we'll 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 some 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 that M would do, given the data D.

Such a machine would come closer to the sort of computers we're familiar with. While the usual Turing Machine is described as a single-purpose device with a single program hard-wired' into its instruction table, the Universal Turing Machine is a multi-purpose device that's 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! 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!

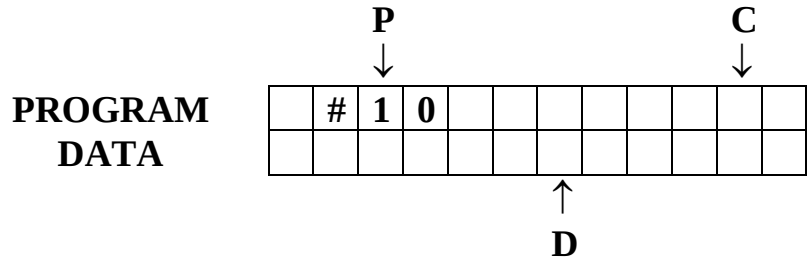
So to support the claim that whatever can be achieved computationally by any real programming language can be achieved by a Turing Machine we'll outline the construction of a Universal Turing Machine. If that does not convince you perhaps you should try to construct a C++ compiler as a Turing Machine!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

We'll construct it as a 2-track, 3-head, 3-direction, 3-character machine which, as we've seen, could be converted to a Standard 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 Standard 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	1L2	1R1
1	0R0	1L3

We write these instructions as binary strings as follows:

	0	**1**
0	1011	111
1	01	10111

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. This is followed by a string of 1's to represent the state, with *n* 1's representing state *n*.

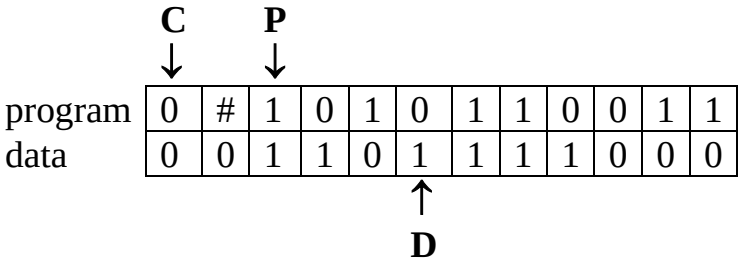
We now combine these into a single list with instructions separated by 0's, and surround the entire string between a pair of #'s, as #1011**0**111**0**01**0**10111#. (The 0 separators are printed here in bold to help you see more clearly the four instructions. When implemented these 0's would be indistinguishable from all other zeros.

How will the Universal Turing Machine interpret this? The first two characters will be the character to be printed and the direction. The very next '0' will be the first separator. Then two more characters and the next '0' will be the next separator, and so on. So there'll be no ambiguity.

When the Universal Turing Machine begins, the heads are placed as follows:

D-head	On the square of 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

Suppose that the tape on the Turing Machine being simulated is: … 0 1 1 0 **.** 1 1 1 1 0 … The 2-track 3-head Universal Turing Machine will start with the following tape:



program	1	0	1	0	1	1	1	#	0	0	0	0
data	0	0	0	0	0	0	0	0	0	0	0	0

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:

	h	**0**	**1**	**#**	
0	D	0N4	1N1		Read 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 to 0 and right again.

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:

	h	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:

	h	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.

	h	0	1	#	
10	P	0L12	1R11	#L12	Copy the state to the
11	D	1L10			left of the left #.

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.

	h	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).

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

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've 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 too.

Putting this altogether we have a Universal Turing Machine with only 20 states.

	h	0	1	#	
0	D	0N4	1N1		Read 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		right to 0 and right again
4	P	0R5	1R6		Read what is to be printed.
5	D	0N7	0N7		Print a 0
6	D	1N7	1N7		or a 1.
7	P	0R8	1R9		Read the direction.
8	D	0L10	1L10		Move the D-head left
9	D	0R10	1R10		or right.
10	P	0L12	1R11	#L12	Copy the state to the
11	D	1L10			left of the left #.
12	P	0L12	1L12	#R13	Reset the Program Pointer.
13	C	0R13	0R14	#L0	Decrement the counter.
14	P	0R15	1R15		Move past
15	P	0R16	1R16		the first
16	P	0R17	1R16		instruction.
17	P	0R18	1R18		Move past
18	P	0R19	1R19		the second
19	P	0R13	1R19	#L20	instruction.

Of course once we convert this to a Standard Turing Machine, with only two characters, one track, one head and no neutral direction, the number of states would probably be many hundreds. But such a Universal Turing Machine does indeed exist.

§ 9.6. Computable Numbers

There are those who seek to adopt a constructivist approach to mathematics. They argue that the uncountability of the real numbers is a myth. Only those real numbers that can be precisely defined, or computed in some way, can be considered to actually exist. And clearly the set of such numbers is countably infinite. "What's the use of having uncountably numbers that can't ever be used, or even precisely defined." they argue.

There's a lot of merit in what they say, but mathematicians have long been unconcerned about whether mathematical objects really exist. They take the pragmatic view that if it's useful, or even convenient, to have certain mathematical objects then they are considered to exist. But why would numbers that can't be computed be at all useful?

Turing defined a **computable real number** as one whose decimal, or binary, expansion can be generated by a Turing Machine. The constructivists would argue that these are the only real numbers that exist and the ability of a Turing Machine to compute the expansion should be

built into the definition of a real number. Thus the real numbers would be countable. “Give me an example of these so-called non-computable real number,” they would ask. Clearly that’s not possible.

Now Turing implied that for a real number to be computable you had to have a Turing Machine that can print out the decimal places of its decimal expansion. Such a process clearly will never terminate, so our convention of having the head move back to its initial position would not be possible. But Turing suggested that for all *n* the Turing Machine that’s generating the decimal expansion would, in finite time, give exactly the first *n* decimal places.

There’s a problem here. Suppose we added 8/330 to 5/66. We should get 0.1000 … or, perhaps, 0.0999 … But, reading the decimal expansions of these numbers, the TM would never know whether the answer is
a tiny bit less than 0.1, or a tiny bit more. So it would never know whether the first decimal place is 0 or 1. Since 8/330 = 0.0242424 … and 5/66 = 0.0757575 … the decimal places that emerge from the TM would be 0.9999 … 9. If this continued for ever, then 0.999 … would be just as acceptable as 1.000 …. But the 1001st decimal place of the second number might be 4 or it might be 6. Only then could we guarantee even the first decimal place of the answer.

The way to solve this problem is to define a computable real number as one where there’s a Turing

Machine such that if a_n is the rational approximation after n steps we can prove that a_n converges to the exact value as $n \to \infty$.

Now, defining a computable number is one thing. Proving all the usual properties of a real number is another. For example we'd want to know that the sum of two computable numbers is computable.

Perhaps we'd need to use a 2-track machine, with the decimal expansion of each number on a separate track. But it can be done fairly elegantly on a single track machine, or even a Standard binary TM provided we write the numbers in binary.

We'll write the binary places of the numbers alternately, and we'll ignore the position of the 'decimal' point. A similar problem existed with the old slide rule that once could be seen sticking out of every engineer's pocket. The slide rule only gave the first three or four significant figures of the answer and the engineer was able to figure out where the decimal place should go. Anyway, a slightly more complicated TM can supply the decimal point.

The tape will contain the binary places in the expansion, alternately. So if $a = 0.a_1a_2a_3 \ldots$ and $b = 0.b_1b_2b_3 \ldots$ then the tape will be as follows:

……	a_1	b_1	a_2	b_2	a_3	b_3	…..

The head will start at the a_1 position. We scan each pair of squares, erasing the first in each pair. If $a_n = 0$ we leave the b_n unchanged and move right.
If $a_n = 1$ and $b_n = 0$ we swap a_n and b_n and move right.
If $a_n = b_n = 1$ we erase both and go back to previous pairs. While ever $b_m = 1$ for $m < n$ we erase it and continue to go back. But as soon as we reach a $b_m = 0$ we change it to 1 and move forward to the next pair.

C	**0**	**1**
0	0R1	0R2
1	0R0	1R0
2	1R0	0L3
3	0L2	

But that doesn't completely solve the problem. The computable number doesn't come to us all nicely written out as a decimal expansion. If we have two computable numbers *a*, *b* we have, by definition, two Turing Machines A, B that generate the expansions. And we can't use them to generate the expansions because they never halt. We need to interleave the machines A, B with the above TM, C. We operate A to print a_1 on our addition track. We then operate B to print b_1 next to it. We Then operate C to perform the addition. We continue with a_2, b_2 etc. At some stage, when we are up to the a_n, b_n pair it may be that A or B need to change a_m or b_m for some $m < n$. We would need to accommodate this.

But if you think *that* is complicated, what about multiplication? Multiplication is incredibly more complicated. While it could be done on a Standard TM we'd be mad to attempt it on anything other than a TM with a 'tape' that has infinitely many rows and columns. If we had the binary expansions of the two numbers on two of the tracks we would then perform 'long multiplication' with the first *n* bits of each number and write the result on an 'output track'. As in normal long multiplication this would require *n* tracks for our working. As we proceeded we may need to rewrite some of the earlier bits on the output track, so at no stage could we be sure of these earlier bits. But the approximations would represent a sequence of rational numbers that converges to the actual value. But, as with addition, we don't get the expansions of our two numbers all written out ready for us. We have to generate them using the associated TM's. Our final machine would have to be an interleaving of those machines with the one we devise to perform the multiplication.

If we were to insist on developing real numbers in this constructionist way, we'd have to face up to all this complexity. A serious course on Turing Machines would have to be taught in every high school before any real use of real numbers was made – certainly before we could begin to study Calculus. And what would we have gained? Intellectual honesty perhaps? We wouldn't have

to allow these ‘off with the fairies’ numbers that ‘don’t exist’.

But mathematics is not about fundamental truths. It’s a made-up world – pure fantasy! Though it’s a world of fantasy it is, on the one hand, fascinating, and on the other hand, extremely useful. Elegance and simplicity are far more important than being concrete.

Of course it is vitally important that our mathematics is logically consistent. We never sacrifice mathematical rigour for some purely fanciful idea. But the standard theory of real numbers is as logically consistent as any other part of mathematics and is far simpler, and far more elegant, than the constructionist’s account. So what if we’ve let in all these numbers that are fantasies? I’m reminded of the parable of the wheat and the tares in the Bible (Matthew 13). Put up with the tares (weeds that look like wheat) until the harvest. Put up with these non-computable numbers which are of no use because it’s too much work to separate them from the computable numbers and, in the end, it doesn’t really matter.

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 an # which locates a '1' under the following conditions. There's 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 it's a much simpler machine.)

Ex 9A5: Construct a Turing Machine on the alphabet {0, 1, #}, where '0' denotes a blank, that 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 HALVE(n) = 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 $n = 1$.

Ex 9A9: In exercise we wrote a program on a 2-head Turing Machine for swapping two numbers. For such a simple task the resulting Standard Turing Machine turned out to be quite large. Construct a Standard Turing Machine to do the same job. There will only be one head, which will start at the beginning of the first number. (Who said two heads are better than one?)
Provide some commentary.

	0	**1**	
0	0R6	0R1	subtract 1 from the left number
1	0R2	1R1	jump over the separator
2	0R3	1R2	move past the end of the second number
3	1L4	1R3	add 1 to the partially relocated first number
4	0L5	1L4	move to right hand end of second number
5	0R0	1L5	move to left hand end of the 1st number

SOLUTIONS FOR CHAPTER 9

Ex 9A1:

	0	**1**
0	0L1	$1L0_R$
$\mathbf{0_R}$	0R0	1R0
1	$0L1_R$	1L2
$\mathbf{1_R}$	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	$1L1_L$	$1L_0$
$\mathbf{1_L}$	0L1	1L1
1	$0R1_R$	1R2
$\mathbf{1_R}$	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	
1	#L2			Put in extra #
2	0L2	1L2	#R3	
3	#R4	#R9	#L14	Read char
4	0R4	1R4	#R5	IF 0
5	0R5	1R5	0R6	copy
6	#L7			and
7	0L7	1L7	#L8	reinstate
8	0L8	1L8	0R3	
9	0R9	1R9	#R10	IF 1
10	0R10	1R10	0R11	copy
11	#L12			and
12	0L12	1L12	#L13	reinstate
13	0L13	1L13	0R3	
14	0L14	1L14	#R15	Reset head

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

In state 0 we move to the last bit. In state 1 we look for the previous 0 and change it to 1, and then reset the head in state 2. However if there is no 0, such as in #111# we change the left # to 1 and put a new # at the left.

(ii) Basically all we need to do is to add a 0 at the end. This can be achieved as follows:

	0	**1**	**#**
0	0R0	1R1	0R1
1	#L2		
2	0L2	1L2	#R3

The only trouble is that zero is represented as ## and this would produce #0#. So we need an extra state to deal with this.

	0	**1**	**#**
0		1R1	#L4
1	0R1	1R1	0R2
2	#L3		
3	0L3	1L3	#R4

We can be very clever and combine states 0 and 2 to get a smaller machine.

	0	**1**	**#**
0	#L2	1R1	#L4
1	0R1	1R1	0R0
2	0L2	1L2	#R3

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		

Ex: 9A9:

Relocating the separator sounds simple but it is quite difficult on an STM. Instead we can leapfrog the first number over the second. Take 1 off the left hand end and transfer it to the right, until you reach the separator. Check that it works if one of the numbers is zero.

	0	1	
0	0R6	0R1	Subtract 1 from the left number
1	0R2	1R1	Jump over the separator
2	0R3	1R2	Move past the end of the second number
3	1L4	1R3	Add 1 to the first number
4	0L5	1L4	Move left to right hand end of 2nd number
5	0R0	1L5	Move left to left hand end of 1st number