Monash University
Faculty of Information Technology
2nd Semester 2023

FIT2014 Theory of Computation
Exercises 7
Turing Machines

SOLUTIONS

Although you may not need to do all the exercises in this Exercise Sheet, it is still important that you attempt all the main questions and a selection of the Supplementary Exercises.

Even for those Supplementary Exercises that you do not attempt seriously, you should still give some thought to how to do them before reading the solutions.

1.

(i) $\underline{\mathbf{a}}\mathbf{aa} \rightarrow \#\underline{\mathbf{a}}\mathbf{a} \rightarrow \#\mathbf{a}\underline{\mathbf{a}} \rightarrow \#\mathbf{aa}\underline{\mathbf{\Delta}} \rightarrow \#\mathbf{a}\underline{\mathbf{a}}\# \rightarrow \#\underline{\#}\mathbf{a}\# \rightarrow \#\underline{\mathbf{a}}\# \rightarrow \#\#\underline{\#} \rightarrow \#\#\#\underline{\mathbf{\Delta}}$ **Accept**

(ii) $\underline{\mathbf{a}}\mathbf{ba} \rightarrow \#\underline{\mathbf{b}}\mathbf{a} \rightarrow \#\mathbf{b}\underline{\mathbf{a}} \rightarrow \#\mathbf{ba}\underline{\mathbf{\Delta}} \rightarrow \#\mathbf{b}\underline{\mathbf{a}} \rightarrow \#\underline{\mathbf{b}}\# \rightarrow \underline{\#}\mathbf{b}\# \rightarrow \#\underline{\mathbf{b}}\# \rightarrow \#\#\underline{\#}$ **CRASH**

(iii) $\underline{\mathbf{b}}\mathbf{aaba} \rightarrow \#\underline{\mathbf{a}}\mathbf{aba} \rightarrow \#\mathbf{a}\underline{\mathbf{a}}\mathbf{ba} \rightarrow \#\mathbf{aa}\underline{\mathbf{b}}\mathbf{a} \rightarrow \#\mathbf{aab}\underline{\mathbf{a}} \rightarrow \#\mathbf{aa}\underline{\mathbf{b}}\# \rightarrow \#\mathbf{a}\underline{\mathbf{a}}\mathbf{b}\# \rightarrow \#\underline{\mathbf{a}}\mathbf{ab}\# \rightarrow \underline{\#}\mathbf{aab}\# \rightarrow \#\underline{\mathbf{a}}\mathbf{aab}\# \rightarrow \#\#\underline{\mathbf{a}}\mathbf{b}\# \rightarrow \#\#\mathbf{a}\underline{\mathbf{b}}\# \rightarrow \#\#\mathbf{ab}\underline{\#} \rightarrow \#\#\mathbf{a}\underline{\mathbf{b}}\# \rightarrow \#\#\underline{\mathbf{a}}\#\# \rightarrow \#\underline{\#}\mathbf{a}\#\# \rightarrow \#\#\underline{\mathbf{a}}\#\# \rightarrow \#\#\#\#\underline{\#} \rightarrow \#\#\#\#\underline{\#}$ **Accept**

(iv) $\underline{\mathbf{a}}\mathbf{babb} \rightarrow \#\underline{\mathbf{b}}\mathbf{abb} \rightarrow \#\mathbf{b}\underline{\mathbf{a}}\mathbf{bb} \rightarrow \#\mathbf{ba}\underline{\mathbf{b}}\mathbf{b} \rightarrow \#\mathbf{babb}\underline{\mathbf{\Delta}} \rightarrow \#\mathbf{babb}\underline{\mathbf{b}} \rightarrow \#\mathbf{ba}\underline{\mathbf{b}}\# \rightarrow \#\mathbf{b}\underline{\mathbf{a}}\mathbf{b}\# \rightarrow \#\underline{\mathbf{b}}\mathbf{ab}\# \rightarrow \underline{\#}\mathbf{bab}\# \rightarrow \#\underline{\mathbf{b}}\mathbf{ab}\# \rightarrow \#\#\underline{\mathbf{a}}\mathbf{b}\# \rightarrow \#\#\mathbf{a}\underline{\mathbf{b}}\# \rightarrow \#\#\mathbf{ab}\underline{\#} \rightarrow \#\#\mathbf{a}\underline{\mathbf{b}}\# \rightarrow \#\#\underline{\mathbf{a}}\#\# \rightarrow \#\underline{\#}\mathbf{a}\#\# \rightarrow \#\#\underline{\mathbf{a}}\#\# \rightarrow \#\#\#\#\underline{\#} \rightarrow \#\#\#\#\underline{\#}$ **Accept**

2.

(i). $\underline{\mathbf{\Delta}} \rightarrow \mathbf{\Delta}\underline{\mathbf{\Delta}}$ **HALT**

(ii). $\underline{\mathbf{a}} \rightarrow \#\underline{\mathbf{\Delta}} \rightarrow \underline{\#}\mathbf{A} \rightarrow \mathbf{a}\underline{\mathbf{A}} \rightarrow \mathbf{aa}\underline{\mathbf{\Delta}} \rightarrow \mathbf{a}\underline{\mathbf{a}}$ **HALT**

(iii). $\underline{\mathbf{a}}\mathbf{a} \rightarrow \#\underline{\mathbf{a}} \rightarrow \#\mathbf{a}\underline{\mathbf{\Delta}} \rightarrow \#\underline{\mathbf{a}}\mathbf{A} \rightarrow \underline{\#}\mathbf{aA} \rightarrow \mathbf{a}\underline{\mathbf{a}}\mathbf{A} \rightarrow \mathbf{a}\#\underline{\mathbf{A}} \rightarrow \mathbf{a}\#\mathbf{A}\underline{\mathbf{\Delta}} \rightarrow \mathbf{a}\#\underline{\mathbf{A}}\mathbf{A} \rightarrow \mathbf{a}\underline{\#}\mathbf{AA} \rightarrow \mathbf{aa}\underline{\mathbf{A}}\mathbf{A} \rightarrow \mathbf{aaa}\underline{\mathbf{A}} \rightarrow \mathbf{aaaa}\underline{\mathbf{\Delta}} \rightarrow \mathbf{aaa}\underline{\mathbf{a}}$ **HALT**

3. In a Finite Automaton, computation can enter a Final State and then leave it again if there is more input to read. A Final State only takes effect (to accept the string) if the FA is in the Final State after reading the entire input string.

By contrast, in a Turing machine, as soon as computation enters an Accept State, it stops and the input is accepted. This happens even if the machine has not read all the input letters yet.

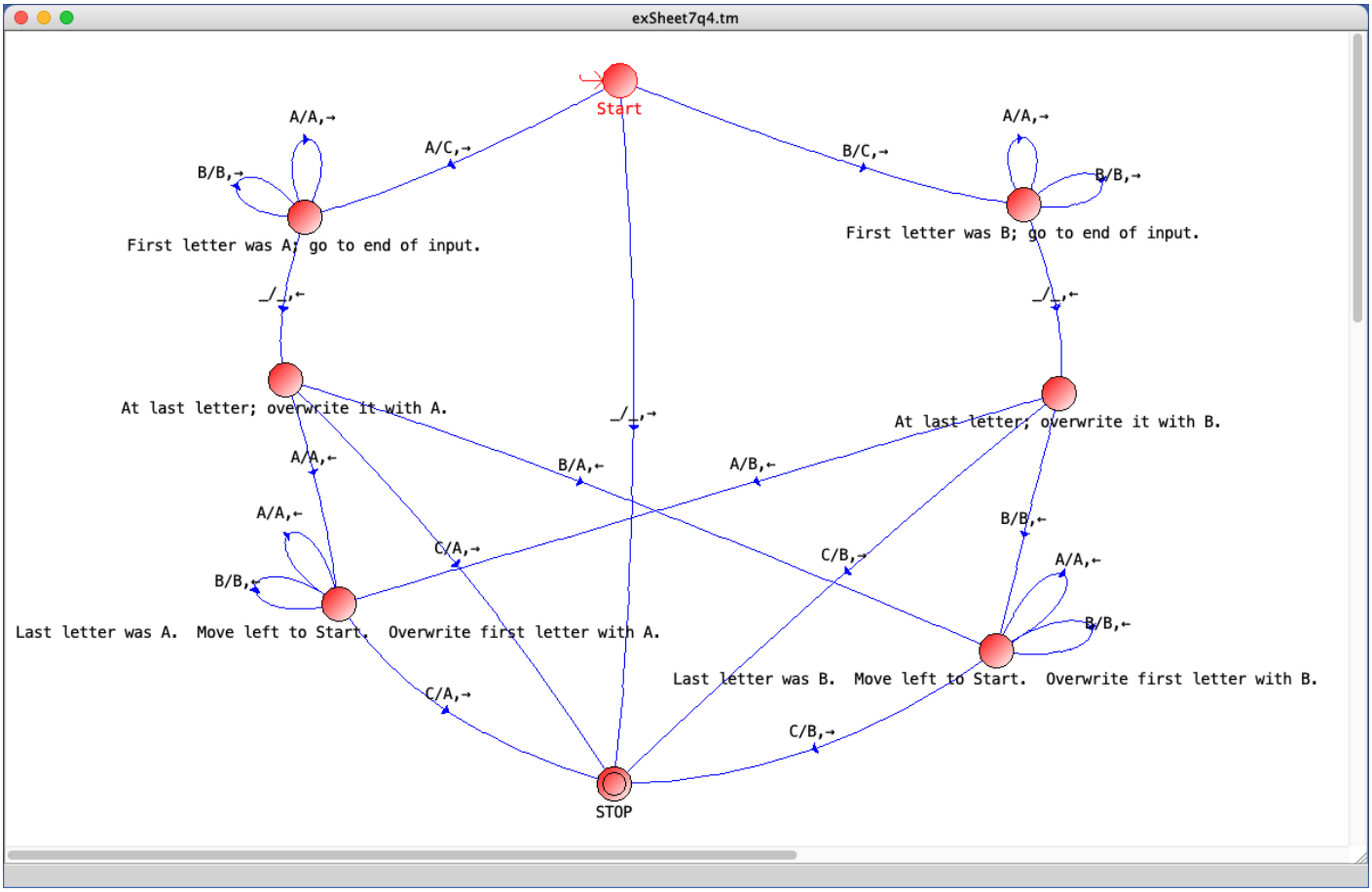
A more minor difference is that, conventionally, FAs are allowed to have multiple Final States whereas Turing machines only have one Accept State. But this is a pretty superficial difference. It is possible to define a different type of Turing machine, in which multiple Accept States are allowed. Such Turing machines can, in fact, be simulated by TMs with just one Accept State,[1] so they don't give anything fundamentally new.

[1] How would you demonstrate this?

4.
(a)

	current state	current symbol	new state	new symbol	direction
/*	Mark first cell with `c` (to avoid crashing off left end later), start moving rightwards. */				
/*	The state we go to depends on what the first letter is. */				
/*	Don't forget the empty string! */				
Start	1	`a`	3	`c`	R
	1	`b`	4	`c`	R
	1	Δ	2	Δ	R
Accept	2				
/*	First letter was `a`. Go to end of input. */				
	3	`a`	3	`a`	R
	3	`b`	3	`b`	R
	3	Δ	5	Δ	L
/*	First letter was `b`. Go to end of input. */				
	4	`a`	4	`a`	R
	4	`b`	4	`b`	R
	4	Δ	6	Δ	L
/*	At last letter. Overwrite it with `a`. */				
	5	`a`	7	`a`	L
	5	`b`	8	`a`	L
	5	`c`	2	`a`	R
/*	At last letter. Overwrite it with `b`. */				
	6	`a`	7	`b`	L
	6	`b`	8	`b`	L
	6	`c`	2	`b`	R
/*	Last letter was `a`. Move left to Start. Overwrite first letter with `a`. */				
	7	`a`	7	`a`	L
	7	`b`	7	`b`	L
	7	`c`	2	`a`	R
/*	Last letter was `b`. Move left to Start. Overwrite first letter with `b`. */				
	8	`a`	8	`a`	L
	8	`b`	8	`b`	L
	8	`c`	2	`b`	R

(b)[2]



(d)

The time is the total of:

- time to get to the blank just after the last input letter: n steps.
- time to get back to the first input letter: n steps.
- after recognising the c marking the first tape cell, one more step to the right as it enters Accept State.

This gives $2n + 1$ steps in total. So $t(n) = 2n + 1$. So the time taken is $O(n)$.

For this TM, the time taken is the same for all inputs of length n. This is rare in general.

[2]Puzzle: Can this Turing machine diagram be drawn on the plane without any crossings? Why or why not?

5.

From	To	Read	Write	Move
1	3	a	#	R
3	3	a	a	R
3	4	b	b	R
4	4	b	b	R
4	5	Δ	Δ	L
5	6	b	#	L
6	7	b	#	L
7	7	a	a	L
7	7	b	b	L
7	8	#	#	R
8	2	#	#	R
8	9	a	#	R
9	9	a	a	R
9	9	b	b	R
9	10	#	#	L
10	7	b	#	L
1	11	b	b	R
11	2	Δ	Δ	R

6.

From	To	Read	Write	Move
1	2	Δ	Δ	R
1	3	a	#	R
3	4	Δ	Δ	L
4	2	#	Δ	R
3	5	a	a	R
5	5	a	a	R
5	6	Δ	Δ	L
6	7	a	Δ	L
7	7	a	a	L
7	1	#	a	R

7.

(i)

Such a TM decides whether to accept or reject based only on what's in the first tape cell, which might be blank (if the input string is the empty string) or the first letter of the input string. So the languages that can be accepted by such a TM are (using regular expression notation, since they are all regular languages):

$$\emptyset \qquad\qquad \varepsilon$$
$$\mathbf{a}(\mathbf{a}\cup\mathbf{b})^* \qquad\qquad \mathbf{a}(\mathbf{a}\cup\mathbf{b})^*\cup\varepsilon$$
$$\mathbf{b}(\mathbf{a}\cup\mathbf{b})^* \qquad\qquad \mathbf{b}(\mathbf{a}\cup\mathbf{b})^*\cup\varepsilon$$
$$(\mathbf{a}\cup\mathbf{b})(\mathbf{a}\cup\mathbf{b})^* \qquad\qquad (\mathbf{a}\cup\mathbf{b})(\mathbf{a}\cup\mathbf{b})^*\cup\varepsilon,$$
$$\text{which is the same as } (\mathbf{a}\cup\mathbf{b})^*.$$

(ii)

These TMs read one character at a time, never re-reading them. Although they may overwrite a character they have read, they always go to the right so the new character is never seen again.

This TM is really just a FA. The class of languages recognised by them is therefore the class of regular languages.

(iii) (This excercise is Sipser ex. 3.13.)

Consider a transition from state p to state q labelled $x \to y, S$, where S denotes Stay Still. If this transition is taken, then there may be subsequent stay-still transitions, taking execution through some sequence of states and changing the character at the current tape cell, possibly several times. All the while, no more input characters are read. Since the TM is deterministic, this sequence of transitions is completely determined. If it leads to acceptance, then we can replace the transition from p to q labelled $x \to y, S$ by one from p to the Accept state labelled x. Otherwise, the sequence ends with a Right step transition, say $x' \to y', R$, to some state q'. Then we can replace the transition from p to q labelled $x \to y, S$ by one from p to q' labelled $x \to y', R$.

Doing this for each transition with direction S gives an equivalent TM in which all transitions have direction R. But this is equivalent to a FA, as we saw in (ii).

(iv)

If you have a TM that crashes when attempting to move off the left-hand end of the tape, you can use it to simulate a TM of the other type as follows.

Create new TM states and transitions that do some preprocessing as follows. Shift the input string one cell to the right, marking the first cell (now vacated by the input string) with a new special symbol, say \$. Then position the Tape Head on the cell to the right of \$, which is the first letter of the input string. Then enter the original TM at its start state, and process the input just as it does. (Everything is happening one cell to the right, but the TM doesn't know that.) This new TM also needs new loop transitions, one at each state of the old TM and each of them labelled by $\$ \to \$, R$. If at any stage the Tape Head reaches the first cell, which means it has just moved Left from what *was* the first cell of the tape of the *original* TM, then it moves Right without changing the \$ and so returns to the second cell of the tape (i.e., the first cell of the original tape) without changing state. So, we have simulated standing still when trying to move off the left end of the tape, using a TM that would crash if we attempted to do so.

Conversely, if we have a TM that stands still when attempting to move off the left end of the tape, then we can use it to simulate one that would crash if it attempted to move left from there. The argument is similar to the above, except that if we are reading the \$ character, then we need to go immediately into a Reject state (rather than just looping at our current state and moving Right, as we did above). Alternatively, if we want to avoid Reject states, we could simply not provide any transitions for the character \$, so that, if the TM finds itself reading \$ — which will only happen if it has moved to the left of the left-hand end of the original tape — then it crashes.

So the class of languages recognised is the same as that for normal TMs.

8.

The main idea is to use the two stacks to represent the Turing machine tape, as follows. The portion of the tape from the start up to, but not including, the position of the Tape Head is in the first stack. The portion of the tape from the Tape Head up to, and including, the last non-blank letter is in the second stack, with the last non-blank letter at the bottom of that stack and the Tape Head position corresponding to the top of the stack. So the letter at the Tape Head position is on the top of the second stack.

To simulate the Turing machine, the 2PDA has a set-up phase where it moves the input string into the second stack, with the last letter of input at the bottom and the first letter of input at the top. The first stack is still empty, and the input tape of the 2PDA is now empty too.

Then, for each Turing machine instruction, we alter the stacks in the 2PDA in a way that simulates the alteration to the tape and the Tape Head position in the TM.

Consider a general TM instruction for transition from state p to state q, which has the form $x \to y, d$, where x and y are letters and $d \in \{\text{Left}, \text{Right}\}$ is the direction in which the Tape Head moves after reading x and writing y.

We will assume that a 2PDA transition $t, u, v \to w, z$ means: if the input letter is t, the top of the first stack is u, and the top of the second stack is v, then we pop u and v from their respective stacks, and push w onto the first stack and z onto the second stack. Any or all of these can be ε, with the same interpretation as in PDAs.

In the 2PDA, the way we translate the TM transition $x \to y, d$ depends on d.

If $d = \text{Right}$, then we have a transition from p to q labelled by $\varepsilon, \varepsilon, x \to y, \varepsilon$. This has the effect of popping x from the top of the second stack and pushing y onto the top of the first, which simulates moving the Tape Head to the Right after overwriting x by y. It is only done if the letter on top of the second stack is x, which is as it should be.

If $d = \text{Left}$, then a little more fiddling is needed. All we want to do is to move the letter at the top of the first stack to the top of the second stack, but we must do so only when the letter at the top of the second stack is x, and x must be replaced by y. We can't just write $\varepsilon, z, x \to \varepsilon, z$, since x was popped before z was put on the second stack, so now has become lost from that stack, whereas we want to change it to y and have it sitting just underneath z. Nor can we just write $\varepsilon, z, \varepsilon \to \varepsilon, z$:

although this does correctly simulate the movement of the tape head one cell to the Left, it will do this whether or not x was sitting atop the second stack, and it will not change x to y. So this is not an exact translation of the TM rules that we are trying to simulate.

We can achieve our desired objective by two transitions.

We create a new state, just for this TM transition, and distinct from all other states. Call it s_{pq}. We first create a transition from p to s_{pq} labelled by $\varepsilon, \varepsilon, x \to \varepsilon, y$, which changes x to y on top of the second stack, and has no other effect on the stacks. But it ensures that we only get to s_{pq} if we have x on the top of the second stack at the beginning of the transition. We then create transitions from s_{pq} to q labelled by $\varepsilon, z, \varepsilon \to \varepsilon, z$, which moves z onto the top of the second stack; there is one such transition from s_{pq} to q for each letter z of the tape alphabet.

This completes the description of how to simulate the TM by a 2PDA.

9.

(a)

$$A_{t,i,\mathbf{a}} \vee A_{t,i,\mathbf{b}} \vee A_{t,i,\Delta}$$

(b)

$$(\neg A_{t,i,\mathbf{a}} \vee \neg A_{t,i,\mathbf{b}}) \wedge (\neg A_{t,i,\mathbf{a}} \vee \neg A_{t,i,\Delta}) \wedge (\neg A_{t,i,\mathbf{b}} \vee \neg A_{t,i,\Delta})$$

(c) This is just the conjunction of the expressions for (a) and (b):

$$(A_{t,i,\mathbf{a}} \vee A_{t,i,\mathbf{b}} \vee A_{t,i,\Delta}) \wedge (\neg A_{t,i,\mathbf{a}} \vee \neg A_{t,i,\mathbf{b}}) \wedge (\neg A_{t,i,\mathbf{a}} \vee \neg A_{t,i,\Delta}) \wedge (\neg A_{t,i,\mathbf{b}} \vee \neg A_{t,i,\Delta})$$

(d)

$$(B_{t,p} \wedge C_{t,i} \wedge A_{t,i,x}) \Rightarrow (B_{(t+1),q} \wedge A_{(t+1),i,y} \wedge C_{(t+1),i+d})$$

10.

If t is the time taken by T for some input, then we are given that the time taken by U to simulate this computation is $\leq t^3$. But we also know that $t \leq n^2$, where n is the input length. Putting these together, the time taken by U to simulate T on an input of length n is $\leq t^3 \leq (n^2)^3 = n^6$.

Extra exercise: what if we wrap big-O notation around the running times in this exercise?

More specifically:

Suppose that T always stops in $O(n^2)$ steps, where n is the length of the input string, and that any computation by any Turing machine M that takes t steps can be simulated by U in $O(t^3)$ steps.

Derive an upper bound, in big-O notation, for the time taken by U to simulate T on an input of length n.

Supplementary exercises

11.

From	To	Read	Write	Move
1	3	b	b	R
1	1	a	a	R
3	4	b	b	R
4	2	b	b	R
3	1	a	a	R
4	1	a	a	R

12.

From	To	Read	Write	Move
1	1	a	a	R
1	2	Δ	Δ	R
1	3	b	b	R
3	1	a	a	R
3	4	b	b	R
3	2	Δ	Δ	R
4	1	a	a	R
4	2	Δ	Δ	R

13.

From	To	Read	Write	Move
1	3	a	#	R
3	3	a	a	R
3	2	Δ	Δ	R
3	5	b	$	L
1	4	b	#	R
4	4	b	b	R
4	5	a	$	L
5	5	a	a	L
5	5	b	b	L
5	5	$	$	L
5	6	#	#	R
6	6	b	b	R
6	6	$	$	R
6	7	a	$	L
7	7	b	b	L
7	7	$	$	L
7	8	#	#	R
8	8	a	a	R
8	8	$	$	R
8	5	b	$	L
8	2	Δ	Δ	R

14.

(i). **$\underline{a}aaba \to A\underline{a}aba \to Aa\underline{a}ba \to Aaa\underline{b}a \to Aaab\underline{a} \to Aaaba\underline{\Delta} \to Aaab\underline{a} \to Aaa\underline{b} \to Aa\underline{a}b \to Aa\Delta\underline{b} \to Aa\Delta b\underline{\Delta} \to Aa\Delta\underline{b} \to Aa\underline{\Delta} \to A\underline{a} \to \underline{A}a \to a\underline{a}$ HALT** $(3-1=2)$

(ii). **$\underline{b}aaa$ CRASH** $(0-3$ not defined$)$

(iii). **$\underline{a}abaa \to A\underline{a}baa \to Aa\underline{b}aa \to Aab\underline{a}a \to Aaba\underline{a} \to Aabaa\underline{\Delta} \to Aaba\underline{a} \to Aab\underline{a} \to Aa\underline{b}a \to Aaba \to A\Delta\underline{b}a \to A\Delta b\underline{a} \to A\Delta ba\underline{\Delta} \to A\Delta b\underline{a} \to A\Delta\underline{b} \to A\underline{\Delta}b \to \underline{A}\Delta b \to \Delta\underline{\Delta}b \to \Delta\Delta\underline{b} \to \Delta\Delta\Delta\underline{\Delta} \to \Delta\Delta\Delta\Delta\underline{\Delta}$ HALT** $(2-2=0)$

15. (This exercise is Sipser ex. 3.14.)

Introduce two new characters, say $ and #, into the tape alphabet. The former, $, will be used to mark both the start and end of the portion of the input tape we have used so far. The latter, #, will be used to help keep track of the Tape Head position, in a manner to be explained below.

For set-up, move all letters of the input string into the queue, with first letter at the head and last letter at the tail, and append \$ to the queue.

Now consider how the Queue Automaton simulates the TM, after this set-up has been done.

Suppose we have a TM transition from state p to state q labelled $x \to y, d$.

If $d =$ Right, we remove (serve) x from the head of the queue and append y to the tail. Then, the letter that was after x in the queue is at the head of the queue, which represents the fact that, in the TM, the Tape Head is now reading that letter.

If $d =$ Left, we would like to remove a letter from the tail and prepend it to the head. But these are not allowable queue operations. So, we achieve the same effect as follows.

We create a set of new states r_{pq}, s_{pqw}, t_{pqw} and u_{pq}, one of each of these for each state p, state q, and (for the middle two) each tape letter w. First, we create a transition from p to r_{pq} which just appends the special letter # to the queue. For now, this letter has the meaning, "the previous letter is the one we now want at the head of the queue". But, when see # at the head, the previous character has already been served, so we need to remember it, which we do using the new states s_{pqw}. These states are used to repeatedly serve the queue and to remember what the last letter we served was.

For each tape letter w except #, we create a transition from r_{pq} to s_{pqw} which serves w from the queue precisely when the input letter is x and the head of the queue is w, and then appends w to the queue.[3]

At the state s_{pqw}, we have a transition to each $s_{pqw'}$, where w' is also in the tape alphabet (except for #). (It is allowed that $w = w'$, in which case we have a loop.) This transition applies when w' is at the head of the queue, and moves it to the tail.

So, as execution moves through the states s_{pqw}, letters are moved from head to tail, with the most recently moved letter always remembered by means of which of these states we are in.

From each state s_{pqw}, we have a transition from s_{pqw} to t_{pqw} which simply moves # from head to tail of queue. Then, for each tape letter w except for #, we have a transition from t_{pqw} to u_{pq} which appends w to the queue. Once we are in state u_{pq}, we know that # is now immediately *ahead* of the letter we want to be at the head of the queue. So we have loop transitions at u_{pq} for each tape alphabet letter except #, moving these letters from head to tail. Finally, we have a transition from u_{pq} to q which simply removes # from the head of the queue, without appending anything. Then, the letter after it becomes the head of the queue, and this is exactly the one we wanted there.

There is a big contrast here between the effort required to simulate a Right step by the TM (just move one letter from head to tail, by one serve and one append) and that required to simulate a Left step (cycling through virtually the whole queue twice). Nonetheless, each step of the TM takes at most about $2k$ steps of the Queue Automaton, where k is the maximum length of TM tape used during the computation.

[3]Note that the state r_{pq} has no transition to deal with # at the head of the queue. But that's ok, as # will never be at the head of the queue when we are in this state.

16.

	current state	current symbol	new state	new symbol	direction
/* Mark first cell (using symbol to remember its contents), start moving rightwards. */					
Start	1	`0`	3	`A`	R
	1	`1`	3	`B`	R
	1	`#`	4	`H`	R
Accept	2				
/* Move to the cell after the `#`. */					
	3	`0`	3	`0`	R
	3	`1`	3	`1`	R
	3	`#`	4	`#`	R
/* This bit indicates whether to flip or not. Remember, in next state. Start going left. */					
	4	`0`	5	Δ	L
	4	`1`	6	Δ	L
/* Delete `#`, go back to start, all bits unchanged; restore first bit. */					
	5	`#`	5	Δ	L
	5	`0`	5	`0`	L
	5	`1`	5	`1`	L
	5	`A`	2	`0`	R
	5	`B`	2	`1`	R
	5	`H`	2	Δ	R
/* Delete `#`, go back to start, flipping bits including first. */					
	6	`#`	6	Δ	L
	6	`0`	6	`1`	L
	6	`1`	6	`0`	L
	6	`A`	2	`1`	R
	6	`B`	2	`0`	R
	6	`H`	2	Δ	R

Here it is in Tuatara: