

1a) To show that a TrM can simulate a TM, we need to provide equivalents for left and right movement. At worst, one left or right movement command from cell i can translate into $\log_2 i$ moves up to the root and then another $\log_2 i$ moves down the other half of the tree. So given the base 2 logarithm of the current index, we know at most how many steps the tree machine would have to move to emulate a single movement of the Turing machine. Then, since every movement in the Turing machine can be replaced by a finite number of movements in the tree machine, we know that the two must be equivalent.

Specifically, we can determine the equivalent series of moves by the following procedures. 'Index' is the binary representation of the starting cell index. Let U represent movement to the parent node, L to the left child, and R to the right child.

```
PROCEDURE TREEIZE-L(index)
IF index matches the regexp 0*10, return U
ELSE
  IF the last digit of index is 1, return UL
  ELSE return U, followed by TREEIZE-L(index DIV 2), followed by R
ENDIF
ENDIF
```

```
PROCEDURE TREEIZE-R(index)
IF index matches the regexp 0*1, return L
ELSE
  IF the last digit of index is 0, return UR
  ELSE return U, followed by TREEIZE-R(index DIV 2), followed by L
ENDIF
ENDIF
```

For instance, when moving left from cell 1000, we have the following derivation (where the numbers in parentheses represent the input to the function): $(1000) \rightarrow U(100)R \rightarrow UU(10)RR \rightarrow UUURR$. When moving right from cell 1011, we have $(1011) \rightarrow U(101)L \rightarrow UU(10)LL \rightarrow UUURLL$.

1b) To show that a Turing machine can simulate a tree machine, we show how to construct a Turing machine T that exactly recognizes the language of the tree machine T' . For each state q_n in T' , add four states $q_n, q_{nU}, q_{nL}, q_{nR}$ to T . Then for each transition $\langle q_m, \alpha \rangle \vdash \langle q_n, X, \beta \rangle$ in T' , add the transition $\langle q_m, \alpha \rangle \vdash \langle q_{nX}, S, \beta \rangle$ to T .

Then add transitions to effect the following (all ϵ/ϵ with respect to the tape): at all times, one tape of T contains the current index that T' would be visiting. Whenever in a state q_nX , T first invokes a subroutine LEFT, UP, or RIGHT (as appropriate) on the current index to calculate where T' would move to. It then invokes a subroutine MOVE to move T to that cell of the input (this is the only way the machine T moves from one cell on the tape to another). The machine then transfers control to state q_n , where the next character is read. Since we know that a Turing machine can calculate the functions $\lfloor i/2 \rfloor$, $2i$, and $2i + 1$, we know that LEFT, UP, and RIGHT must exist. The subroutine MOVE can, at the most primitive level, simply move to the left edge of the tape and then advance as many spaces as needed. (A more sophisticated version could calculate the change in position required at each movement and act appropriately.)

2) The formal transition table is as follows (without error transitions):

State	Input	ToState	Output	Move	Explanation
0	0	1	\$	R	Insert edge protector
0	B	2	\$	R	(If empty tape)
1	0	1	0	R	Move to end of input
1	B	2	0	R	Put a 0 there
2	B	3	1	L	Put a 1 at the end of the input
3	0	3	0	L	Move to the beginning
3	1	3	1	L	
3	\$	4	\$	R	We're now looking at the first input character
4	B	4	B	R	Find the first non-blank character
4	1	11	1	L	If it's a one...
4	2	11	2	L	... or a two, move on to the next phase
4	0	5	B	R	Otherwise, erase it and start another cycle
5	0	5	0	R	Find first one or two
5	1	6	2	R	Replace 1's with twice as many 2's
5	2	8	1	R	Replace 2's with twice as many 1's
6	1	6	1	R	Go to end of tape...
6	2	6	2	R	
6	B	7	2	L	... and deposit a 2 there.
7	2	7	2	L	Go back and look for another one
7	1	6	2	R	Change it to a 2 and loop again
7	0	10	0	L	But if we find a 0, we're done with this cycle
7	B	4	B	R	
8	1	8	1	R	Go to end of tape...
8	2	8	2	R	
8	B	9	1	L	... and deposit a 1 there.
9	1	9	1	L	Go back and look for another two
9	2	8	1	R	Change it to a 1 and loop again
9	0	10	0	L	But if we find a 0, we're done with this cycle
9	B	4	B	R	
10	0	10	0	L	Find the first blank, looking leftward...
10	B	4	B	R	... and start the main loop again
11	B	11	B	L	Find the beginning of the tape
11	\$	12	\$	R	We're scanning the left-most blank
12	B	12	B	R	Move over all the initial blanks
12	1	13	0	R	Start changing all the 1's or 2's to 0's
12	2	13	0	R	
12	0	13	0	R	
13	1	13	0	R	Change the rest of the 1's or 2's to 0's
13	2	13	0	R	
13	0	13	0	R	
13	B	14	B	L	Start shifting tape left
14	0	15	B	L	Absorb 0 at end of tape
15	0	15	0	L	Move to right-most blank
15	B	16	0	L	Write a zero there
15	\$	20	\$	S	Or skip to the end if there aren't any
16	B	16	B	L	Find the left edge of the tape
16	\$	17	\$	R	
17	B	12	B	R	If blanks remain, shift left again
17	0	18	0	R	Otherwise, we're on the last pass
18	0	18	0	R	Find the end of the zeroes
18	B	19	B	L	
19	0	20	B	L	Grab the zero
20	0	20	0	L	Find the beginning of the tape...
20	\$	acc	0	S	... and overwrite the protector with the final zero

For every element of $Q \times \{\$, B, 0, 1, 2\}$ not listed in the table above, there is a transition to the state $q_r e j$ that leaves the tape unchanged and does not move the head.

At a high level, we have the following algorithm. We start by inserting an edge protector at the beginning of the tape so that later we can move back to the beginning of the tape without worrying about falling off the tape. We then write a 1 at the end of the input to represent 2^0 . Now, for each 0 in the input, we erase it and double the number of digits following the zeroes, replacing, for instance, two 1's with four 2's, or eight 2's with sixteen 1's. We alternate between 1's and 2's as a way of 'marking off' the symbols that have already been duplicated. Once all of the input has been erased, we shift the answer to the left edge and change all the digits to zeroes. Finally, we remove the edge protector, leaving the machine in the state $q_{acc}0^{2^n}$.

For example, the evaluation of q_0 (i.e. on an empty tape) proceeds as follows:

$$q_0 \vdash \$q_2 \vdash q_3\$1 \vdash \$q_41 \vdash q_11\$1 \vdash \$q_{12}1 \vdash \$0q_{13} \vdash \$q_{14}0 \vdash q_{15}\$ \vdash q_{20}\$ \vdash q_{acc}0$$

As another example, the evaluation of q_0000 proceeds as follows:

$$\begin{aligned} q_0000 \vdash & \$q_100 \vdash \$0q_10 \vdash \$00q_1 \vdash \$000q_2 \vdash \$00q_301 \vdash \$0q_3001 \vdash \$q_30001 \vdash q_3\$0001 \vdash \$q_40001 \vdash \$Bq_5001 \\ & \vdash \$B0q_501 \vdash \$B00q_51 \vdash \$B002q_6 \vdash \$B00q_722 \vdash \$B0q_7022 \vdash \$Bq_{10}0022 \vdash \$q_{10}B0022 \vdash \$Bq_40022 \\ & \vdash \$BBq_5022 \vdash \$BB0q_522 \vdash \$BB01q_82 \vdash \$BB012q_8 \vdash \$BB01q_921 \vdash \$BB011q_81 \vdash \$BB0111q_8 \\ & \vdash \$BB011q_911 \vdash \$BB01q_9111 \vdash \$BB0q_91111 \vdash \$BBq_901111 \vdash \$Bq_{10}B01111 \vdash \$BBq_401111 \\ & \vdash \$BBBq_51111 \vdash \$BBB2q_6111 \vdash \$BBB21q_611 \vdash \$BBB211q_61 \vdash \$BBB2111q_6 \vdash \$BBB211q_712 \\ & \vdash \$BBB2112q_62 \vdash \$BBB21122q_6 \vdash \$BBB2112q_722 \vdash \$BBB211q_7222 \vdash \$BBB21q_71222 \\ & \vdash \$BBB212q_6222 \vdash^* \$BBB21222q_6 \vdash \$BBB21222q_722 \vdash^* \$BBB2q_7122222 \vdash^* \$BBB222222q_6 \\ & \vdash \$BBB22222q_722 \vdash^* \$BBq_7B2^8 \vdash \$BBBq_42^8 \vdash \$BBq_{11}B2^8 \vdash^* q_11\$BBB2^8 \vdash \$q_{12}BBB2^8 \\ & \vdash^* \$BBBq_{12}2^8 \vdash \$BBB0q_{13}2^7 \vdash^* \$BBB0^8q_{13} \vdash \$BBB0^7q_{14}0 \vdash \$BBB0^6q_{15}0 \vdash^* \$BBq_{15}B0^7 \\ & \vdash \$Bq_{16}B0^8 \vdash^* q_{16}\$BB0^8 \vdash \$q_{17}BB0^8 \vdash \$Bq_{12}B0^8 \vdash \$BBq_{12}0^8 \vdash^* \$BB0^8q_{13} \vdash \$BB0^7q_{14}0 \\ & \vdash \$BB0^6q_{15}0 \vdash^* \$Bq_{15}B0^7 \vdash \$q_{16}B0^8 \vdash q_{16}\$B0^8 \vdash \$q_{17}B0^8 \vdash \$Bq_{12}0^8 \vdash \$B0q_{13}0^7 \vdash^* \$B0^8q_{13} \\ & \vdash \$B0^7q_{14}0 \vdash \$B0^6q_{15}0 \vdash^* \$q_{15}B0^7 \vdash q_{16}\$0^8 \vdash \$q_{17}0^8 \vdash \$0q_{18}0^7 \vdash \$0^8q_{18} \vdash \$0^7q_{19}0 \vdash \$0^6q_{20}0 \\ & \vdash^* q_{20}\$0^7 \vdash q_{acc}0^8 \end{aligned}$$

3a) From the text ([LP] 235), we know that $mult(x, y)$, which returns the product of x and y , is a primitive recursive function. Then $fact(x)$ is a primitive recursive function:

$$fact(x) = \begin{cases} succ(0), & \text{if } x = 0; \\ mult(succ(P_1^2(x-1, fact(x-1))), P_2^2(x-1, fact(x-1))), & \text{otherwise.} \end{cases}$$

3b) (This is basically translating [LP]'s notation into the notation of the notes, since remainder(m,n) is given on page 236.)

Define the predecessor function as:

$$pred(x) = \begin{cases} 0, & \text{if } x = 0; \\ P_1^2(x-1, pred(x-1)), & \text{otherwise.} \end{cases}$$

Then define monus (floored subtraction, in a sense) as:

$$monus(x, y) = \begin{cases} P_1^1(x), & \text{if } y = 0; \\ pred(P_3^3(x, y-1, monus(x, y-1))), & \text{otherwise.} \end{cases}$$

Now introduce an is-zero predicate:

$$iszero(x) = \begin{cases} succ(0), & \text{if } x = 0; \\ 0, & \text{otherwise.} \end{cases}$$

Then define some more useful predicates:

$$\begin{aligned} geq(x, y) &= iszero(monus(P_2^2(x, y), P_1^2(x, y))) \\ leq(x, y) &= monus(succ(0), geq(P_1^2(x, y), P_2^2(x, y))) \\ eql(x, y) &= mult(leq(P_1^2(x, y), P_2^2(x, y)), geq(P_1^2(x, y), P_2^2(x, y))) \end{aligned}$$

Define a helper function:

$$r2(m, n) = \begin{cases} 0, & \text{if } n = 0; \\ mult(monus(succ(0), eql(P_3^3(n, m-1), r2(n, m-1))), pred(P_1^3(n, m-1), r2(n, m-1))), \\ \quad succ(P_3^3(n, m-1), r2(n, m-1))), & \text{otherwise.} \end{cases}$$

Now we can define remainder:

$$remainder(m, n) = mult(monus(1, iszero(P_1^2(m, n))), r2(P_1^2(m, n), P_2^2(m, n)))$$

4) In the simulation of a NTM by a TM given in the notes, the third tape contains all possible ‘choices’ for the computation. As we execute each of these choices, we mark the choice off (on a second track)—1 if it reached an accepting state and 0 otherwise. Instead of stopping when we find a single accepting computation, we run the machine until all possible computations have been run. Then, we accept if every computation is either accepting or the prefix of an accepting computation.

By this procedure, we can completely simulate a forall TM by a normal deterministic TM. Therefore the set of languages recognized by the forall TM is exactly the recursive enumerable languages.