# Problem Set 5: Turing Machines
# CS254, Fall 2019, Anna Rafferty

You are welcome to talk to others about the problems on this problem set, but you should **read and attempt each problem prior do discussing it with others.** The write up must be your own work. That means that **you should write your answers by yourself, without looking at notes from sessions with collaborators, and you must be able to explain any answer you write down.** You should list anyone you collaborated with on the collaboration form; remember, that form is due when you turn in the final problem for a problem set.

**Due dates:** Two problems are due on Monday, October 14, and the remaining problems are due on Wednesday, October 16. [Note: Typo in due dates corrected - please always check the course schedule and/or ask if the due dates don't match what I say in class or are inconsistent between days of the week and numeric dates.] As for all homeworks, if a problem is due on a specific day it must be turned in by 9PM on that day. See homework handout (a copy is posted on Moodle) for more specific homework policies. Failing to name your files properly may result in a loss of points and potentially receiving no credit for that problem; please name your files properly! Please do not remove homework from the Hand-In folder after you've turned it in, as leaving it there provides a record for when it was turned in if there is any dispute later.

1. (Modified from Sipser.) Consider the scramble operation on a language $L$:

$$\text{scramble}(L) = \{w \mid w \text{ is the anagram of some string } x \in L\}$$

An anagram of a string $x$ is a string $y$ that contains exactly the same characters as $x$, each occurring the same number of times as in $x$, but in any order. For example, if $L = \{cat\}$, $\text{scramble}(L) = \{act, atc, cat, cta, tac, tca\}$.

(a) Assume that $\Sigma = \{0, 1\}$. Show that if $L$ is regular, scramble$(L)$ is context free. *Hint: Construct a PDA. Use the ability to read from both the stack and the input to non-deterministically read through the string in all possible orders. Your description of the PDA can be a high level description (not a tuple or a diagram) if you wish, but it must be precise enough that I could create the tuple/diagram if I wished. You should argue why your construction is correct.*

**Solution:** As suggested by the hint, we'll construct a PDA that recognizes scramble$(L)$ given that $L$ is a regular language. Since $L$ is a regular language, there must exist a DFA $D = (Q, \Sigma, \delta, q_0, F)$ for it. In our PDA, our states will be the same as in the DFA except that we'll add to addition states: a new start state $q_0'$ and a new accept state $q_f$. For the new start state $q_0'$, the only transition will be $\delta(q_0', \epsilon, \epsilon) = (q_0, \$)$. That is, our new start state just pushes a \$ to mark the bottom of the stack and transitions to the original start state. For each state, we'll have three types of (non-deterministic) transitions for each character:

1. Pop $\epsilon$ from the stack, read the next character in the input string, and put it on the stack. Remain in the same current state.

2. Pop $\epsilon$ from the stack, read the next character $c$ in the input string, and transition to $\delta(q, c)$, where $q$ is the current state.

3. Pop the top character $c$ from the stack, read an $\epsilon$ from the input string, and transition to $\delta(q, c)$, where $q$ is the current state.

For the accept states for the PDA, the only accept state will be $q_f$, and $q_f$ will have incoming transitions from each of the accept states in the DFA $D$ (i.e., from each $f \in F$). These transitions will read $\epsilon$ from the input and pop \$ from the top of the stack.

We now need to justify why this machine accepts all and only strings in scramble$(L)$. First, if an input $w$ is accepted by this machine, the input must have been fully read and the machine has ended in $q_f$, the only accept state. The only way to reach $q_f$ is by transitioning on a \$ from the stack from an accept state, and no additional transitions are possible after $q_f$ is reached. This means that the machine accepts only given an empty stack, and prior to finishing the input/emptying the stack, the machine must have come from an accept state in the original DFA. Examining the transitions, they either push a character from the input onto the stack, remaining in the same state, or transition according to the transitions of the original DFA based on reading a character from the input string or reading a character from the top of the stack (but not both simultaneously). Thus, each character in the original string is used for exactly one transition in the states of the original DFA, and these transitions begin in the start state and end in an accept state if the machine accepts. This means that some permutation of the characters in the input $w$ would be accepted by the original DFA $D$, so by definition $w \in \text{scramble}(L)$.

Second, we need to argue that if a string $w = w_1 \dots w_n$ is in scramble$(L)$, then the machine we've constructed accepts it. First, note that if $w = \epsilon \in \text{scramble}(L)$, then the start state of $D$ was an accept state. The constructed machine will accept $\epsilon$ by pushing on the \$ and transitioning to $q_0$, then popping off the \$ and transitioning to $q_f$.

We now consider cases where $w \neq \epsilon$ and $w \in \text{scramble}(L)$. That means there's some permutation of $w$, call it $w' = w'_1 \dots w'_n$, such that $w' inL$. We want to argue that one of the non-deterministic paths through our PDA will transition through the states of the DFA in exactly the order of $w'$. We'll show via induction on the state of the machine after having completed $i$ instances of type 2 and 3 transitions (above). We want to show that for any $i$, there's a non-deterministic path where the state is $\hat{\delta}(q_0, w'_1 \dots w_i)$ and the stack contains only 0s or only 1s (or neither 0s nor 1s), plus the bottom of stack marker.

Base case: We'll show that when $i = 1$ (i.e., we've done exactly one transition of type 2 or 3 in this path), there's a non-deterministic path where the stack contains only 0s or only 1s (or neither 0s nor 1s), plus the bottom of stack marker, and the state is $\hat{\delta}(q_0, w'_1) = \delta(q_0, w'_1)$. If the first character of $w'$ is the same as the first character of $w$, then option (2) above means that one of the non-deterministic paths matches $w'$ through the first character. On this path, the state will be $\hat{\delta}(q_0, w'_1)$ and the stack will contain only the bottom of stack marker.

If the first characters are not the same, then let $j$ be the index of the furthest left character in $w$ that matches the first character of $w'$. The non-deterministic path that follows option (1) above $j-1$ times will results in all of the first $j-1$ (identical) characters on the stack, and can then follow a transition of type (2) to match the initial character. This takes us to $\hat{\delta}(q_0, w'_1)$, and the stack will contain $j-1$ of one character, the bottom of stack marker, and nothing else.

Inductive case: Assume that we've done exactly $k$ transitions of type 2 or 3 in some path, the stack contains only one of 0s or 1s (or neither), and we're in state $\hat{\delta}(q_0, w'_1 \dots w'_k)$. Consider the identity of character $w'_{k+1}$. If it's the same as the top of the stack symbol or the next symbol of the input, then we can take a single transition of type (2) or (3) above, transition to $\delta(\hat{\delta}(q_0, w'_1 \dots w'_k), w'_{k+1}) = \hat{\delta}(q_0, w'_1 \dots w'_{k+1})$, and not add any symbols to the stack, meaning that the property of the stack containing only 0s, 1s, or neither is preserved.

Alternatively, if $w'_{k+1}$ is not equal to either the top of the stack symbol or the next symbol of the input, then either the stack has no 0s or 1s in it, or it has the same symbol as the current input. That means we can take transitions of type (1) until we reach a character in the input that matches $w'_{k+1}$; we know we must eventually reach such a character because $w'$ is a permutation of $w$, and the fact that the stack contains only 0s or only 1s (or neither) means that the character we want can't be buried somewhere in the stack. When we reach that character, we will still have only 0s or 1s in the stack, since we have either the same symbol we started with or we started with no 0s or 1s, and we can then take a transition of type (2) to match $w'_{k+1}$. As above, this transitions us to $\delta(\hat{\delta}(q_0, w'_1 \dots w'_k), w'_{k+1}) = \hat{\delta}(q_0, w'_1 \dots w'_{k+1})$.

Thus, we've shown by induction that for any $i$, there's a path that takes the machine to $\hat{\delta}(q_0, w'_1 \dots w'_i)$ after performing exactly $i$ transitions of type 2 or 3. Taking $i = n$, we know that we will have followed $n$ type 2 or 3 transitions, so we will have read the entire string through the PDA and off of the stack (since each type 2 or 3 transition consumes a character and there are only $n$ characters to consume), and we have that the machine will be in $\hat{\delta}(q_0, w'_1 \dots w'_n)$. Because $w'_i \in L$, this means the machine is in one of the original accept states from the DFA, and the stack is empty, so it can transition to $q_f$ by popping a \$. Thus, this machine accepts $w$ for any $w \in L$.

*Note for grading: It's important that both of these parts are argued for your construction, but your argument may be slightly less rigorous.*

(b) Assume that $\Sigma = \{0, 1, 2\}$. Show that if $L$ is regular, it is possible that scramble$(L)$ is not context free.

**Solution:** Consider $L((012)^*)$. This language contains sequence of 012, repeated zero or more times. Thus scramble$(L) = \{w \mid w$ contains equal numbers of 0s, 1s, and 2s$\}$. We've shown that $\{0^n 1^n 2^n \mid n \geq 0\}$ is not context free. We could adapt that same argument so show that scramble$(L)$ is not context free: Assume scramble$(L)$ is context free. Then there exists a pumping length $p$ such that for all strings $w \in \text{scramble}(L)$, $|w| \geq p$, we should be able to write $w = uvxyz$ such that $|vxy| \leq p$, $|vy| > 0$, and for all $i$, $uv^i xy^i z \in L$. Consider the string $w = 0^p 1^p 2^p$. This string is of length $3p$ and is in $L$, but as we showed in class, any way of splitting it into parts $u, v, x, y, z$ and then setting $i = 0$ results in an unequal number of 0s, 1s, and 2s, meaning the resulting string is not in scramble($L$). This is a contradiction of the pumping lemma for CFLs, so scramble($L$) is not a CFL.

2. Sometimes, it can be helpful to see a Turing machine in action. Here's code for a machine that performs binary addition:

```
; Binary addition - adds two binary numbers
; Input: two binary numbers, separated by a single space, eg '100 1110'

0 _ _ r 1
0 * * r 0
1 _ _ l 2
1 * * r 1
2 0 _ l 3x
2 1 _ l 3y
2 _ _ l 7
3x _ _ l 4x
3x * * l 3x
3y _ _ l 4y
3y * * l 3y
4x 0 x r 0
4x 1 y r 0
4x _ x r 0
4x * * l 4x    ; skip the x/y's
4y 0 1 * 5
4y 1 0 l 4y
4y _ 1 * 5
4y * * l 4y    ; skip the x/y's
5 x x l 6
5 y y l 6
5 _ _ l 6
5 * * r 5
6 0 x r 0
6 1 y r 0

7 x 0 l 7
7 y 1 l 7
7 _ _ r halt
7 * * l 7
```

This code halts with the sum of two binary numbers as the only thing on the tape. You can run this code here: http://morphett.info/turing/turing.html. (Either copy and paste it, or alternatively, click Load an Example Program and load the binary addition program.) Note that the simulator is a variant of the TMs we introduced in class - it has infinite space to the left and right of the input - this doesn't change anything else about how it works.

(a) Try stepping through the execution of the machine with several sample inputs and try to understand what's happening in the code and how that results in the desired output; reading through the webpage of the simulator will help with understanding the syntax. For example, each line is a transition rule, with the first element being the current state, the second the current symbol on the tape, the third the new symbol to write, the fourth the direction to move, and the fifth the new state to transition to. As you go through the code, write comments for each state that describe its purpose. Make sure you understand why the x's and y's are being written. For this part, turn in your commented version of the code (semicolons start comments). Just as in when writing comments for a program in Python or some other language, your comments should make the structure of the computation and the reason for what's being done make more sense - don't just repeat exactly what a line does (e.g., "writes an a, moves to the left, and stays in state 15 if an a is read" would not be a useful comment).

(b) Write a pseudocode algorithm (or step by step English) for what the Turing machine is doing.

(c) Modify the code (or write your own from scratch) so that it takes as input three binary numbers, each separated by a space. It should accept (enter a state named halt-accept) if the third number is the sum of the first two, and reject (enter a state named halt-reject) if the third number is not the sum of the first two. Include comments in your modified code.

**Solution:** Part a. Commented version:

```
; Binary addition - adds two binary numbers
; Input: two binary numbers, separated by a single space, eg '100 1110'


; Scan to the right over the initial number
0 _ _ r 1
0 * * r 0

; Scan more to the right over the second number in the input,
; then move one left to end with the head on the last digit of the second number
1 _ _ l 2
1 * * r 1

; Delete the last digit of the second number,
; and remember it by switching to 3x (if it was a 0)
; and switching to 3y (if it was a 1). This allows it
; to then move to the left using the 3x/y state to add the
; digit to the initial number
2 0 _ l 3x
2 1 _ l 3y
2 _ _ l 7

; Scan left until the last digit of the first number is hit
; 3x vs 3y is for remembering if we'll be adding a 0 or a 1
3x _ _ l 4x
3x * * l 3x
3y _ _ l 4y
3y * * l 3y

; Mark that the last digit has been added to, using x vs y
; to know if the final version of that last digit will be a
; 0 or a 1 - this is used so we know which place to add the next
; number from the second number to. (i.e., the place that will
; never change again in the sum are xs and ys)
4x 0 x r 0
4x 1 y r 0
4x _ x r 0
4x * * l 4x    ; skip the x/y's
; Add a 1 to the last digit, taking care of any carrying that's necessary
4y 0 1 * 5
4y 1 0 l 4y
4y _ 1 * 5
4y * * l 4y    ; skip the x/y's

; Move to the end of the digits in the first number after a number has been
; added, leaving the head on the last digit of that number (i.e., the last
; spot not yet replaced with an x/y
5 x x l 6
5 y y l 6
5 _ _ l 6
5 * * r 5
```

```
; Mark that the last digit has been added to, using x vs y
; to know if the final version of that last digit will be a
; 0 or a 1
6 0 x r 0
6 1 y r 0

; Go back over the sum (after the second number has been added to it)
; and turn the xs and ys back into 0s and 1s. Halt when all x/ys have
; been converted
7 x 0 l 7
7 y 1 l 7
7 _ _ r halt
7 * * l 7
```

Part b.

```
    while there are some digits in the second number:
         curDigit = last (furthest right) digit of the second number
         erase curDigit from the end of the second number (write a blank in that position)
         move head to the last digit of the first number
         if curDigit is a 1:
            adding = True
            while adding:
                if digit at the spot where the head is equals 1:
                    change it to a 0 and move left; we're carrying a 1 up the place values
                else:
                    change the current square to a 1
                    set adding to False
         move back to the rightmost digit in the first number and
         change it to an x if it's a 0, a y if it's a 1
    change all x's to 0s and y's to 1s, and then halt
```

Part c.

```
; Scan to the right over the initial number
0 _ _ r 1
0 * * r 0

; Scan more to the right over the second number in the input,
; then move one left to end with the head on the last digit of the second number
1 _ _ l 2
1 * * r 1

; Delete the last digit of the second number,
; and remember it by switching to 3x (if it was a 0)
; and switching to 3y (if it was a 1). This allows it
; to then move to the left using the 3x/y state to add the
; digit to the initial number
2 0 _ l 3x
2 1 _ l 3y
2 _ _ l 7

; Scan left until the last digit of the first number is hit
; 3x vs 3y is for remembering if we'll be adding a 0 or a 1
3x _ _ l 4x
3x * * l 3x
3y _ _ l 4y
```

```
3y * * l 3y

; Mark that the last digit has been added to, using x vs y
; to know if the final version of that last digit will be a
; 0 or a 1 - this is used so we know which place to add the next
; number from the second number to. (i.e., the place that will
; never change again in the sum are xs and ys)
4x 0 x r 0
4x 1 y r 0
4x _ x r 0
4x * * l 4x    ; skip the x/y's
; Add a 1 to the last digit, taking care of any carrying that's necessary
4y 0 1 * 5
4y 1 0 l 4y
4y _ 1 * 5
4y * * l 4y    ; skip the x/y's

; Move to the end of the digits in the first number after a number has been
; added, leaving the head on the last digit of that number (i.e., the last
; spot not yet replaced with an x/y
5 x x l 6
5 y y l 6
5 _ _ l 6
5 * * r 5

; Mark that the last digit has been added to, using x vs y
; to know if the final version of that last digit will be a
; 0 or a 1
6 0 x r 0
6 1 y r 0

; Go back over the sum (after the second number has been added to it)
; and turn the xs and ys back into 0s and 1s. Move to comparing to
; the third number when all x/ys have been converted
7 x 0 l 7
7 y 1 l 7
7 _ _ r allZeros
7 * * l 7

; Then add on a bit at the end to match numbers
; The non-numbered states below just take care of the case of leading
; 0s; state 8 is where the checking picks up once any
; leading 0s have been handled
allZeros 0 _ r allZeros
allZeros 1 1 r atLeastOne1
allZeros _ _ r checkAllZeros

checkAllZeros _ _ r checkAllZeros
checkAllZeros 0 0 r checkZerosInAnswer
checkAllZeros 1 1 r halt-reject

checkZerosInAnswer 0 0 r checkZerosInAnswer
checkZerosInAnswer 1 1 r halt-reject
checkZerosInAnswer _ _ r halt-accept

atLeastOne1 1 1 r atLeastOne1
atLeastOne1 0 0 r atLeastOne1
atLeastOne1 _ _ r removeLeadingZeros
```

```
removeLeadingZeros _ _ r removeLeadingZeros
removeLeadingZeros 0 _ r removeLeadingZeros
removeLeadingZeros 1 1 l moveToFarLeft

moveToFarLeft _ _ l moveToFarLeft
moveToFarLeft 1 1 l findLeftOfFirstNumber
moveToFarLeft 0 0 l findLeftOfFirstNumber

findLeftOfFirstNumber * * l findLeftOfFirstNumber
findLeftOfFirstNumber _ _ r 8

; Look for the beginning of the first number; when found, remember the digit
; and match it in the second number
8 _ _ r 8
8 1 x r 9a
8 0 x r 9c
8 y y r halt-accept

; Look for a the beginning of the third number to match a 1
9a * * r 9a
9a _ _ r 9b
9b _ _ r 9b
9b * * * 10a

; Look for a the beginning of the third number to match a 0
9c * * r 9c
9c _ _ r 9d
9d _ _ r 9d
9d * * * 10b

; Check if the character in the third number matches: looking for a 1
10a 0 0 r halt-reject
10a 1 y l 11
10a _ _ r halt-reject
10a y y r 10a

; Check if the character in the third number matches: looking for a 0
10b 1 1 r halt-reject
10b 0 y l 11
10b _ _ r halt-reject
10b y y r 10b

; Find the leftmost character in first number, and return to number checking in state 8
11 * * l 11
11 x x r 8
```

3. Let $L = \{ww^R \mid w \in \{0,1\}^*\}$ - that is, the language of even length palindromes. Design a Turing machine that decides $L$. Describe your Turing machine at a moderate level: similar to Sipser's descriptions in chapter 3, but slightly more specific about the algorithmic implementation, including some discussion of the states. For instance, while Sipser says things like "find the end of the input", for this one problem I would like you to describe how that's actually done (when does the machine know when to stop? is it using states to remember anything?).

**Solution:** This Turing machine will repeatedly scan to match symbols on either end of the input.

1. First, if the input is blank when it begins (i.e., it starts on a blank square or if assuming a left side of tape symbol, when it moves one right it is on a blank square), then it accepts.
2. Otherwise, it moves to the first square of the input and repeats the following:

(a) Read the current symbol, write a blank to the current square, and transition to a "looking for 0" or "looking for 1" state based on whether the symbol was a 0 or a 1.

(b) Scan right, repeatedly writing the same symbol it reads, until it hits a blank square. Move left, and transition to a new state "matches 0" or "matches 1" based on whether it is looking for a 0 or a 1.

(c) If the symbol on the current square doesn't match the symbol it's looking for (i.e., it's in "matches 0" and reads a 1 or a blank), reject. Otherwise, write a blank to this spot and transition to a scanning left state.

(d) In the scanning left state, keep moving left, writing the same symbol that is read, until it hits a blank. When a blank is reached, go right one square and transition to the same state as at the start of this loop. Repeat from the start.

4. Sipser 3.8b. Give only a high level description (similar to the style of Sipser's descriptions).

**Solution:** On input string $w$:

- Scan the tape and mark the first 0 that has not been marked. If no 0 is found, go to step 5. Otherwise, move the head back to the left side of the tape.
- Scan the tape and mark the first 0 that has not been marked. If no 0 is found, reject. Otherwise, move the head back to the left side of the tape.
- Scan the tape and mark the first 1 that has not been marked. If no 1 is found, reject.
- Move the head back to the start of the tape and repeat from step 1.
- Move the head back to the left side of the tape. Scan the tape to see if any unmarked 1s remain. If none are found, accept. Otherwise, reject.

5. Sipser 3.13. Prove both parts of your answer correct.

**Solution:** This machine is not equivalent to an ordinary Turing machine. It's exactly equivalent to a DFA. To prove this, we'll show how to transform any machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ with stay put instead of left into a DFA, as well as how to transform any DFA into a RS-TM. We first modify $M$ into a slightly easier form, where none of these changes affect the language recognized by $M$. First, we add a new symbol so that $M$ never *writes* blanks; instead, it writes this new symbol, and the transition function is extended to behave as if it read a blank when this symbol is read. Additionally, when $M$ transitions into $q_{rej}$ or $q_{acc}$ we constrain it to move right, rather than stay in place.

We design the DFA $D = (Q', \Sigma', \delta', q_0', F)$. We let $Q' = Q$, $\Sigma' = \Sigma$, and $q_0' = q_0$. Then we set up $\delta'(q, a)$ as follows. If $q = q_{acc}$ or $q = q_{rej}$, then for all $a$, $\delta'(q, a) = q$.

If $q \neq q_{acc}$ and $q \neq q_{rej}$, we then add in transitions for state-symbol pairs $(q, a)$ where the machine $M$ will never move right after reading $a$ in state $q$. For these pairs, we set $\delta'(q, a) = q_{rej}$. We detect these pairs by checking through a sequence of next states and symbols written based on the transition function. Specifically, if $\delta(q, a) = (r, b, S)$, then we check the output of $\delta(r, b)$. We repeat this, each time looking at the transition from the new state and the output symbol written until one of two things happens. If we get to a transition that moves right, then we know that $M$ eventually moves right after reading $a$ in state $q$. If we check a sequence of $|Q| \times |\Gamma| + 1$ state-symbol pairs and we haven't found a stay, we know $M$ will never move right after reading $a$ in state $q$. There must be a repeat pair by the time we've checked that many pairs (pigeonhole principle) so if it has only stayed put after checking that many, we know that it will never move.

Finally, for all other state-symbol pairs, we'll use the same process to determine the next state upon reading symbol $a$. Specifically, $\delta'(q, a) = p$ if $p$ is the state that $M$ enters when it first moves right after reading $a$ while in state $q$ (this is the state in the first triple with an R for the direction in the sequence of up to $|Q| \times |\Gamma|$ triples described above).

We then define $F = \{q_{acc}\} \cup \{q \mid \hat{\delta}(q, \sqcup^n) = q_a \text{ for some } n\}$. This machine accepts exactly when $M$ accepts: it is in the same state as $M$ would be after reading all input (or in state reject if $M$ would at some point loop while staying put), and potentially reading one or more blanks off the end of the tape.

Since we can transform a RS-TM into a DFA, we know this machine does not accept any languages that are not regular.

Next, we show that it accepts all the regular languages. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, construct a TM that has the same set of states, and transitions $\delta'(q, a) = (p, a, R)$, where $\delta(q, a) = p$. Additionally, create an accept state $q_{acc}$ and a reject state $q_{rej}$. Add transitions $\delta'(f, \sqcup) = (q_{acc}, \sqcup, R)$ for $f \in F$, and $\delta'(g, \sqcup) = (q_{rej}, \sqcup, R)$ for $g \notin F$. Then this machine exactly mimics the DFA: it reads the characters, switching states with each character of the input, and when it reaches the end of the input, it transitions to accept if the DFA ends in an accept state and reject if the DFA ends in a non-accept state. Thus, it will accept exactly the same set of strings as the DFA.

Thus, since we have shown equivalence in both directions, the RS-TM accepts exactly the class of regular languages and is not as powerful as a standard TM.

6. Suppose that we extend Turing machines so that the transition function $\delta$ is a function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R, S\}$, where $\delta(p, a) = (q, b, S)$ means if you are in state $p$ and scanning symbol $a$ on the tape, then overwrite the $a$ with a $b$, enter state $q$, and leave the tape head exactly where it is. (That is, in addition to being able to move left or right, we've added the ability to stay in one place.) What languages can be accepted by these variant Turing machines? Prove your answer correct.

**Solution:** These machines can accept the exact same set of languages as regular Turing machines. To show this, we must show (a) that we can simulate a regular TM using a RLS-TM (right-left-stay TM), and (b) that we can simulate a RLS-TM with a regular TM.

(a) Direction 1: If we have a regular TM, we can simulate it on a RLS-TM by keeping the exact same definition. This machine accepts the same language.

(b) Direction 2: If we have a RLS-TM, we can simulate it on a regular TM as follows. The new transition function $\delta'$ is the same as the original transition function $\delta$ with modifications on the stay states. For any $\delta(q, a) = (p, b, S)$, we add a new state $stay-p$ . Then $\delta'(q, a) = (stay-p, b, R)$, $\delta(stay-p, \cdot) = (p, \cdot, L)$. This means we may have up to $2|Q|$ states in the new machine. This machine closely mimics the behavior of the original RLS-TM by moving right and then left (rewriting the character originally on the square to the right in the same place) and ending up in the same state as if the machine had stayed in the same spot. Since both the tape and state are the same as if the machine had stayed, this machine accepts the same language as the original machine.