HW4CSE105W25: Sample solutions

CSE105W25 Team

Due: February 20th at 5pm, via Gradescope

In this assignment,

You will work with context-free languages and their representations. You will also practice analyzing, designing, and working with Turing machines. You will explore recognizable and decidable languages.

Resources: To review the topics for this assignment, see the class material from Weeks 5 and 6. We will post frequently asked questions and our answers to them in a pinned Piazza post.

Reading and extra practice problems: Sipser Chapters 2 and 3. Chapter 2 exercises 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.9, 2.11, 2.12, 2.13, 2.16, 2.17. Chapter 3 exercises 3.1, 3.2, 3.5, 3.8.

Assigned questions

1. **Regular and nonregular languages and context-free grammars (CFG)** (6 points): On page 7 of the week 4 notes, we have the following list of languages over the alphabet $\{a, b\}$

$$\{a^n b^n \mid 0 \leq n \leq 5\} \quad \{b^n a^n \mid n \geq 2\} \quad \{a^m b^n \mid 0 \leq m \leq n\}$$
$$\{a^m b^n \mid m \geq n+3, n \geq 0\} \quad \{b^m a^n \mid m \geq 1, n \geq 3\}$$
$$\{w \in \{a,b\}^* \mid w = w^{\mathcal{R}}\} \qquad \{ww^{\mathcal{R}} \mid w \in \{a,b\}^*\}$$

(a) (*Graded for completeness*) [1] Pick one of the regular languages and design a context-free grammar that generates it. Briefly justify your grammar by describing the role of each of the rules and connecting it to the intended language and referencing relevant definitions.

[1] This means you will get full credit so long as your submission demonstrates honest effort to answer the question. You will not be penalized for incorrect answers. To demonstrate your honest effort in answering the question, we expect you to include your attempt to answer *each* part of the question. If you get stuck with your attempt, you can still demonstrate your effort by explaining where you got stuck and what you did to try to get unstuck.

Solution: Consider the language

$$\{b^m a^n \mid m \geq 1, n \geq 3\}$$

Note that this language is regular because the regular expression bb^*aaaa^* describes it. A CFG that generates it is defined as

$$(\{S\}, \{a, b\}, \{S \to bS \mid Sa \mid baaa\}, S)$$

Starting from the start variable S, each step in the derivations in the grammar can substitute one S with bS, Sa, or $baaa$. The only rule that leads to termination of the derivation, $S \to baaa$, makes sure that there is at least one b followed by three a's as required by the language. The other two rules make sure that we can add more b's in front or add more a's at the end of the string before termination so that the b's always come before the a's. Therefore, the CFG generates the chosen language $\{b^m a^n \mid m \geq 1, n \geq 3\}$.

(b) (*Graded for completeness*) Pick one of the nonregular languages and design a context-free grammar that generates it. Briefly justify your grammar by describing the role of each of the rules and connecting it to the intended language and referencing relevant definitions.

Solution: Consider the language

$$\{ww^{\mathcal{R}} \mid w \in \{a, b\}^*\}$$

A CFG that generates it is defined as

$$(\{S\}, \{a, b\}, \{S \to 0S0 \mid 1S1 \mid \varepsilon\}, S)$$

Starting from the start variable S, each step in the derivations in the grammar can substitute one S with $0S0$, $1S1$, or ε, i.e. we can add one 0 on both sides of S, or one 1 on both sides of S, or terminate the derivation. Therefore, each string in the language generated by the CFG will have even length (because characters are added in pairs) and will be palindromes (because we build the string up from the middle, and at each step ensure that the characters in corresponding positions away from the middle agree).

2. **General constructions for context-free languages** (15 points):

In class in week 5, we described several general constructions with PDAs and CFGs, leaving their details to homework. In this question, we'll fill in these details. The first constructions help us prove that the class of regular languages is a subset of the class of context-free languages. The other construction allows us to make simplifying assumptions about PDAs recognizing languages.

(a) (*Graded for completeness*) When we first introduced PDAs we observed that any NFA can

be transformed to a PDA by not using the stack of the PDA at all. Suppose a friend gives you the following construction to formalize this transformation:

> Given a NFA $N = (Q, \Sigma, \delta_N, q_0, F)$ we define a PDA M with $L(M) = L(N)$ by letting $M = (Q, \Sigma, \Sigma, \delta, q_0, F)$ where $\delta(\ (q, a, b)\ ) = \delta_N(\ (q, a)\ )$ for each $q \in Q$, $a \in \Sigma_\varepsilon$ and $b \in \Sigma_\varepsilon$.

For each of the six defining parameters for the PDA, explain whether it's defined correctly or not. If it is not defined correctly, explain why not and give a new definition for this parameter that corrects the mistake.

Solution: The set of states, input alphabet, stack alphabet, start state, and the set of accept states are defined correctly. However, the transition function is incorrect, because the output type is incorrect. For the output of our PDA transition function, we are expecting an element from $\mathcal{P}(Q \times \Gamma_\varepsilon)$ (where $\Gamma = \Sigma$ in this construction), but now the output would be an element from $\mathcal{P}(Q)$. A correct definition would be:

$$\delta((q, a, b)) = \begin{cases} \delta_N((q, a)) \times \{\varepsilon\} & \text{if } q \in Q,\ a \in \Sigma_\varepsilon,\ b = \varepsilon \\ \emptyset & \text{otherwise} \end{cases}$$

(b) (*Graded for correctness*) [2] In the book on page 107, the top paragraph describes a procedure for converting DFAs to CFGs:

> You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \to \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable ofthe grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

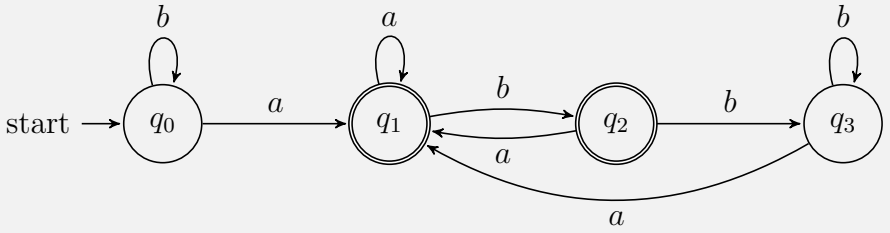
Use this construction to get a context-free grammar generating the language

$$\{w \in \{a, b\}^* \mid w \text{ has at least one } a \text{ and does not end in } bb\}$$

by (1) designing a DFA that recognizes this language and then (2) applying the construction from the book to convert the DFA to an equivalent CFG. A complete and correct submission will include the state diagram of the DFA, a brief justification of why it recognizes the language, and then the complete and precise definition of the CFG that results from applying the construction from the book to this DFA. *Ungraded bonus: take a sample string in the language and see how the computation of the DFA on this string translates to a derivation in your grammar.*

[2] This means your solution will be evaluated not only on the correctness of your answers, but on your ability to present your ideas clearly and logically. You should explain how you arrived at your conclusions, using mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported. Your goal should be to convince the reader that your results and methods are sound.

Solution: A DFA that recognizes this language:



The meaning of each state is the following:

- q_0: We have only seen b's so far. If the computation on a string ends in this state, it means that the string does not contain a and thus should be rejected.
- q_1: We have just read an a. If the computation on a string ends in this state, it means that the string has at least one a, and ends in a which means it does not end in bb, and thus we should accept the string.
- q_2: We have just read a b following an a. If the computation on a string ends in this state, it means that the string has at least one a, and ends in ab which means it does not end in bb, and thus we should accept the string.
- q_3: We have just read at least two b's following an a. If the computation on a string ends in this state, it means that the string has at least one a and ends in bb, and thus we should reject it.

By the above explanation, we can see that all strings in the language $\{w \in \{a,b\}^* \mid w \text{ has at least one } a \text{ and does not end in } bb\}$ (i.e. strings that has at least one a and ends in a or ab) are accepted by the DFA, and we reject all other strings (i.e. strings that have no a or end in bb). Therefore, the DFA recognizes the given language.

Following the construction in the book, we can create a CFG defined as $(\{R_0, R_1, R_2, R_3\}, \{0,1\}, T, R_0)$ where T is a set of rules defined as:

$$
\begin{aligned}
R_0 &\to bR_0 \mid aR_1 \\
R_1 &\to aR_1 \mid bR_2 \mid \varepsilon \\
R_2 &\to aR_1 \mid bR_3 \mid \varepsilon \\
R_3 &\to bR_3 \mid aR_1
\end{aligned}
$$

Notice that the rules $R_1 \to \varepsilon$ and $R_2 \to \varepsilon$ correspond to q_1 and q_2 being accepting states in the DFA above.

Ungraded: Let's use the string bab as an example. We will see that the computation of the DFA on bab corresponds to a derivation of bab in the grammar above in that

the derivation produces the characters in the string from left-to-right, with our intermediate representation then having the variable corresponding to the current state of the computation in the rightmost position. The computation of the DFA on *bab* is the sequence of states

$$q_0, q_0, q_1, q_2$$

The derivation of *bab* in the grammar is

$$R_0 \Rightarrow bR_0 \Rightarrow baR_1 \Rightarrow babR_2 \Rightarrow bab$$

The computation of the DFA starts at the start state q_0. Likewise, the derivation also starts with the start variable (which is the corresponding variable R_0). When we read a b, we're right where we started at q_0, correlating with the rule $R_0 \to bR_0$ and the first step in derivation is $R_0 \Rightarrow bR_0$. When we read an a, we transition to q_1, correlating with the rule $R_0 \to aR_1$, so we take another step in derivation as $R_0 \Rightarrow bR_0 \Rightarrow baR_1$. Finally, when we read a b, we transition to q_2. Likewise, our CFG tells us we can use the rule $R_1 \to bR_2$, so our derivation becomes $R_0 \Rightarrow bR_0 \Rightarrow baR_1 \Rightarrow babR_2$. Finally, we have no more characters to read and because we are in an accept state, we know the string *bab* is accepted by this DFA, that is to say, in the language. In our CFG, we can use the rule $R_2 \to \varepsilon$ so the derivation $R_0 \Rightarrow bR_0 \Rightarrow baR_1 \Rightarrow babR_2 \Rightarrow bab\varepsilon$ ends with string *bab* which has no variable, and thus must be in the language this CFG generates.

(c) Let $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_1, F_1)$ be a PDA and let $q_{new}, r_{new}, s_{new}$ be three fresh state labels (i.e. $Q_1 \cap \{q_{new}, r_{new}, s_{new}\} = \emptyset$) and let # be a fresh stack symbol (i.e. $\# \notin \Gamma_1$). We define the PDA M_2 as

$$(Q_2, \Sigma, \Gamma_2, \delta_2, q_{new}, \{s_{new}\})$$

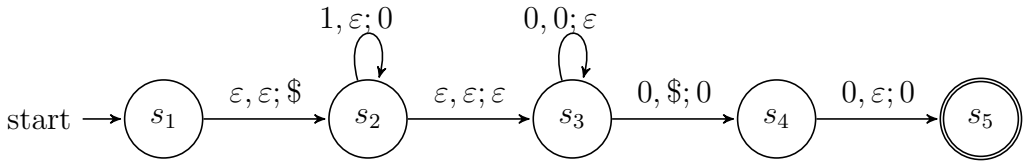
with $Q_2 = Q_1 \cup \{q_{new}, r_{new}, s_{new}\}$ and $\Gamma_2 = \Gamma_1 \cup \{\#\}$ and $\delta_2 : Q_2 \times \Sigma_\varepsilon \times \Gamma_{2\varepsilon} \to \mathcal{P}(Q_2 \times \Gamma_{2\varepsilon})$ given by

$$\delta_2(\ (q,a,b)\ ) = \begin{cases} \{(q_1, \#)\} & \text{if } q = q_{new}, a = \varepsilon, b = \varepsilon \\ \delta_1(\ (q,a,b)\ ) & \text{if } q \in Q_1 \setminus F_1, a \in \Sigma_\varepsilon, b \in \Gamma_{1\varepsilon} \\ \delta_1(\ (q,a,b)\ ) & \text{if } q \in F_1, a \in \Sigma, b \in \Gamma_{1\varepsilon} \\ \delta_1(\ (q,a,b)\ ) & \text{if } q \in F_1, a = \varepsilon, b \in \Gamma_1 \\ \delta_1(\ (q,a,b)\ ) \cup \{(r_{new}, \varepsilon)\} & \text{if } q \in F_1, a = \varepsilon, b = \varepsilon \\ \{(r_{new}, \varepsilon)\} & \text{if } q = r_{new}, a = \varepsilon, b \in \Gamma_1 \\ \{(s_{new}, \varepsilon)\} & \text{if } q = r_{new}, a = \varepsilon, b = \# \\ \emptyset & \text{otherwise} \end{cases}$$

for each $q \in Q_2$, $a \in \Sigma_\varepsilon$, and $b \in \Gamma_{2\varepsilon}$.

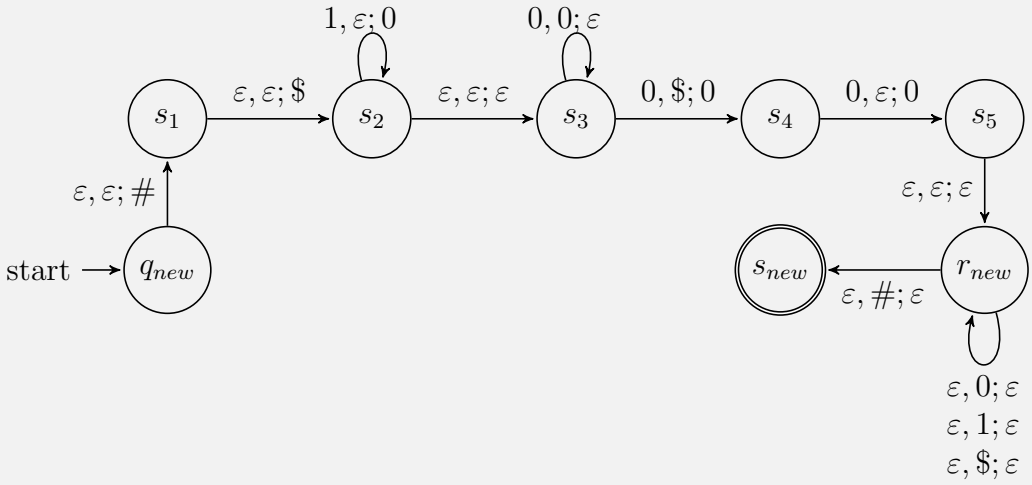
In this question, we'll apply this construction for a specific PDA and use this example to extrapolate the effect of this construction.

i. (*Graded for correctness*) Consider the PDA M_1 with input alphabet $\{0, 1\}$ and stack alphabet $\{0, 1, \$\}$ whose state diagram is



Draw the state diagram for the PDA M_2 that results from applying the construction to M_1. Also, give an example string of length 4 that is accepted by both M_1 and M_2 and justify your choice by describing an accepting computation for each of the PDAs on your input string.

Solution: The state diagram for the PDA M_2 is the following:



Example string of length 4 accepted by both M_1 and M_2: 1000.

An accepting computation of M_1 on 1000 (processes the whole string and ends in an accepting state):

1. start from s_1 with an empty stack
2. push \$ onto the stack and transit to s_2 (current stack content: \$)
3. read 1 from input, push 0 onto the stack and stay in s_2 (current stack content: \$0)
4. spontaneously move to s_3
5. read 0 from input, pop 0 from the stack, and stay in s_3 (current stack content: \$)
6. read 0 from input, pop \$ from the stack, push 0 onto the stack, and move to s_4 (current stack content: 0)

7. read 0 from input, push 0 onto the stack, and move to s_5 (current stack content: 00)

An accepting computation of M_2 on 1000 (processes the whole string and ends in an accepting state):

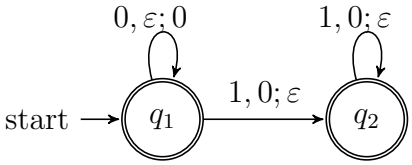
1. start from q_{new} with an empty stack
2. push # onto the stack and transit to s_1 (current stack content: #)
3. push \$ onto the stack and transit to s_2 (current stack content: #\$)
4. read 1 from input, push 0 onto the stack and stay in s_2 (current stack content: #\$0)
5. spontaneously move to s_3
6. read 0 from input, pop 0 from the stack, and stay in s_3 (current stack content: #\$)
7. read 0 from input, pop \$ from the stack, push 0 onto the stack, and move to s_4 (current stack content: #0)
8. read 0 from input, push 0 onto the stack, and move to s_5 (current stack content: #00)
9. spontaneously move to r_{new}
10. pop a 0 from the stack and stay in r_{new} (current stack content: #0)
11. pop a 0 from the stack and stay in r_{new} (current stack content: #)
12. pop a # from the stack and move to s_{new} (current stack empty)

ii. (*Graded for completeness*) Compare $L(M_1)$ and $L(M_2)$. Are these sets equal? Does your answer depend on the specific choice of M_1? Why or why not?

Solution: $L(M_1)$ and $L(M_2)$ are equal. Here, $L(M_1) = L(M_2) = \{1^m 0^{m+2} \mid m \geq 0\}$. More generally, no matter how we choose M_1, we get $L(M_1) = L(M_2)$. Loosely speaking, what M_2 does is to follow M_1's computation and "empty the stack" before a string is accepted. More specifically, for any $s \in L(M_1)$, there exists a computation of M_1 on s that processes the whole string and ends in one of the accept states in M_1, and thus there exists a computation of M_2 on s that processes the whole string and reaches r_{new}. In r_{new}, the stack content can be popped until there is only one # symbol at the bottom of the stack that was pushed when transitioning from q_{new} to the original start state of M_1. Then, from

r_{new}, we can pop out the # from stack and end the computation in s_{new} which is the accept state of M_2. Therefore, s will be accepted by M_2. This shows that every string accepted by M_1 is also accepted by M_2. On the other hand, for any string $t \notin L(M_1)$, there does not exist a computation of M_1 on t that processes the whole string and ends in one of the accept states in M_1, and thus there does not exist a computation of M_2 on t that can reach r_{new} and thus ends in s_{new} which is the only accept state of M_2. Therefore, t will be rejected by M_2. This shows that every string rejected by M_1 will also be rejected by M_2. Therefore, $L(M_1) = L(M_2)$ no matter what M_1 we choose.

iii. (*Graded for completeness*) Consider the PDA N with input alphabet $\{0, 1\}$ and stack alphabet $\{0, 1\}$ whose state diagram is



Remember that the definition of set-wise concatenation is: for languages L_1, L_2 over the alphabet Σ, we have the associated set of strings

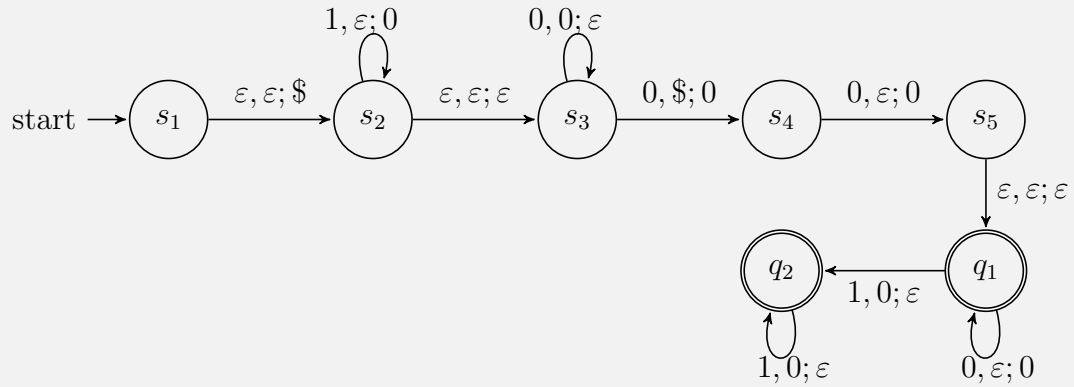
$$L_1 \circ L_2 = \{w \in \Sigma^* \mid w = uv \text{ for some strings } u \in L_1 \text{ and } v \in L_2\}$$

In class, we discussed how extrapolating the construction that we used to prove that the class of regular languages is closed under set-wise concatenation by drawing spontaneous transitions from the accepting states in the first machine to the start state of the second machine doesn't work. Use the example of M_1 and N to prove this by showing that

$$L(M_1) \circ L(N)$$

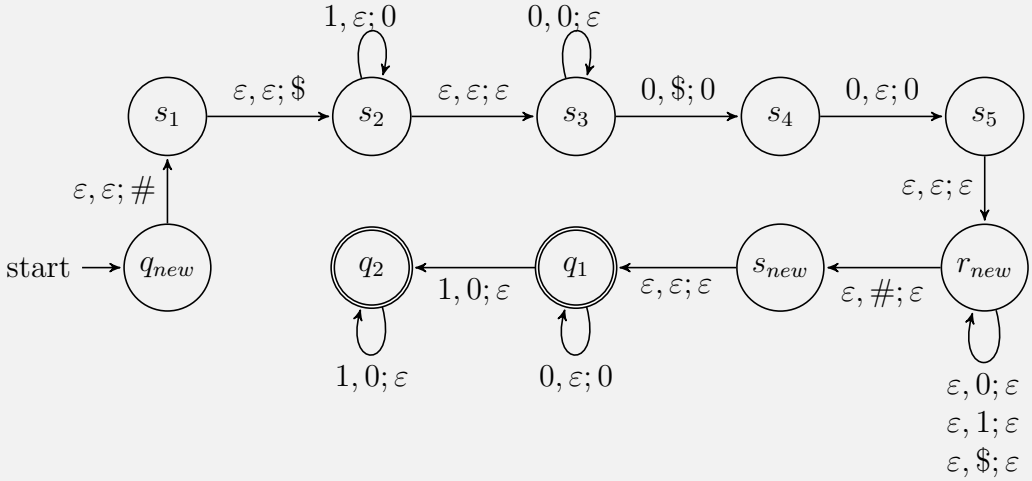
is **not** the language recognized by the machine results from taking the two machines M_1 and N, setting the start state of M_1 to be the start state of the new machine, setting the set of accepting states of N to be the set of accepting states of the new machine, and drawing spontaneous arrows from the accepting states of M_1 to the start state of N. Then, describe the language recognized by the machine that results from taking the two machines M_2 and N, setting the start state of M_2 to be the start state of the new machine, setting the set of accepting states of N to be the set of accepting states of the new machine, and drawing spontaneous arrows from the accepting states of M_2 to the start state of N. Use this description to explain why we used the construction of M_2 from M_1 and how this construction could be used in a proof of the closure of the class of context-free languages under set-wise concatenation.
A complete response will give an example string that witnesses that $L(M_1) \circ L(N)$ is not equal to the language recognized by the PDA resulting from the wrong construction (described above) **and** the state diagram of the PDA that results from applying that construction to M_2 and N (instead of M_1), with a brief justification about why that approach works.

Solution: The state diagram of the new machine P we get following the construction with M_1 and N is the following, where the input alphabet is $\{0,1\}$ and the stack alphabet is $\{0,1,\$\}$:



We can get that $L(M_1) = \{1^m 0^{m+2} \mid m \geq 0\}$ and $L(N) = \{0^m 1^n \mid m \geq n \geq 0\}$. Therefore, $L(M_1) \circ L(N) = \{1^m 0^{m+2} 0^j 1^k \mid m \geq 0,\ j \geq k \geq 0\} = \{1^m 0^p 1^k \mid m, k \geq 0,\ p \geq m + k + 2\}$. However, because the stack has two 0's left at s_5, we get $L(P) = \{1^m 0^{m+2} 0^j 1^k \mid m \geq 0,\ j+2 \geq k \geq 0\} = \{1^m 0^p 1^k \mid m, k \geq 0,\ p \geq m+k\}$. Consider the string 0011 that is accepted by P. However, $0011 \notin L(M_1) \circ L(N)$. Therefore, $L(P) \neq L(M_1) \circ L(N)$.

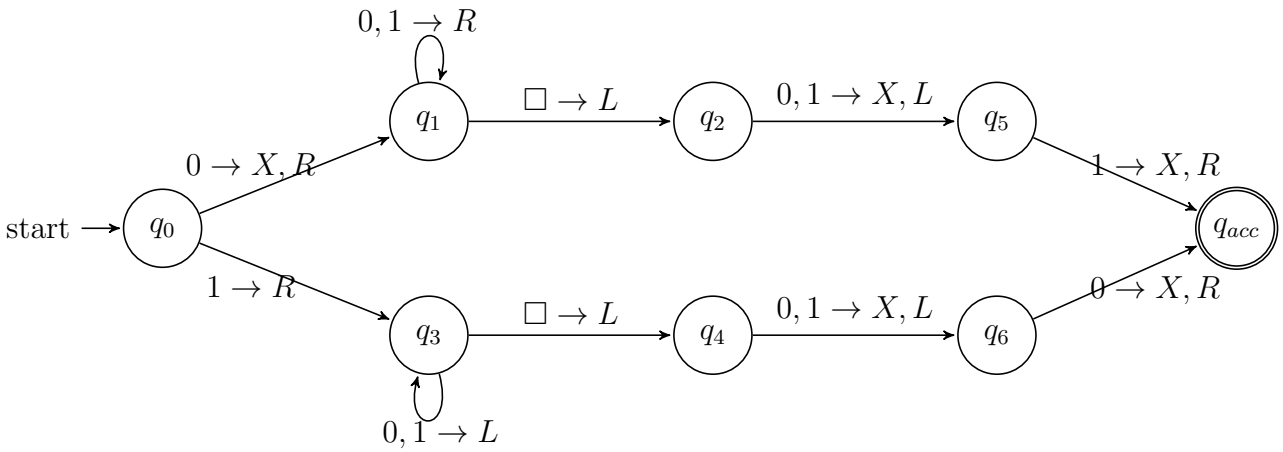
However, let's try this construction with M_2 and N. The state diagram of the new machine Q we get following the construction is the following, where the input alphabet is $\{0,1\}$ and the stack alphabet is $\{0,1,\#,\$\}$:



This works out! We have that $L(Q) = L(M_2) \circ L(N)$ because M_2 clears the stack before we proceed to the simulation of N. This way, there is no remaining 0's on the stack before entering q_1; we're starting with a clean slate from q_1 with an empty stack, and from there we can simulate the computation of N as is.

3. **Turing machines** (9 points):

Consider the Turing machine T over the input alphabet $\Sigma = \{0, 1\}$ with the state diagram below (the tape alphabet is $\Gamma = \{0, 1, X, \square\}$). Convention: we do not include the node for the reject state $qrej$ and any missing transitions in the state diagram have value $(qrej, \square, R)$



(a) (*Graded for correctness*) Specify an example string w_1 of length 4 over Σ that is **accepted** by this Turing machine, or explain why there is no such example. A complete solution will include either (1) a precise and clear description of your example string and a precise and clear description of the accepting computation of the Turing machine on this string or (2) a sufficiently general and correct argument why there is no such example, referring back to the relevant definitions.

To describe a computation of a Turing machine, include the contents of the tape, the state of the machine, and the location of the read/write head at each step in the computation.

Hint: In class we've drawn pictures to represent the configuration of the machine at each step in a computation. You may do so or you may choose to describe these configurations in words.

Solution: Consider the string 0010. Below is the accepting computation of T on 0010:

$q_0 \downarrow$						
0	0	1	0	$\sqcup$	$\sqcup$	$\sqcup$
	$q_1 \downarrow$					
X	0	1	0	$\sqcup$	$\sqcup$	$\sqcup$
		$q_1 \downarrow$				
X	0	1	0	$\sqcup$	$\sqcup$	$\sqcup$
			$q_1 \downarrow$			
X	0	1	0	$\sqcup$	$\sqcup$	$\sqcup$
				$q_1 \downarrow$		
X	0	1	0	$\sqcup$	$\sqcup$	$\sqcup$
				$q_2 \downarrow$		
X	0	1	0	$\sqcup$	$\sqcup$	$\sqcup$
		$q_5 \downarrow$				
X	0	1	X	$\sqcup$	$\sqcup$	$\sqcup$
			$q_{acc} \downarrow$			
X	0	X	X	$\sqcup$	$\sqcup$	$\sqcup$

(b) (*Graded for correctness*) Specify an example string w_2 of length 3 over Σ that is **rejected** by this Turing machine or explain why there is no such example. A complete solution will include either (1) a precise and clear description of your example string and a precise and clear description of the rejecting computation of the Turing machine on this string or (2) a sufficiently general and correct argument why there is no such example, referring back to the relevant definitions.

Solution: Consider the string 000. Below is the rejecting computation of T on 000:

$q_0 \downarrow$					
0	0	0	$\sqcup$	$\sqcup$	$\sqcup$
	$q_1 \downarrow$				
X	0	0	$\sqcup$	$\sqcup$	$\sqcup$
		$q_1 \downarrow$			
X	0	0	$\sqcup$	$\sqcup$	$\sqcup$
			$q_1 \downarrow$		
X	0	0	$\sqcup$	$\sqcup$	$\sqcup$
		$q_2 \downarrow$			
X	0	0	$\sqcup$	$\sqcup$	$\sqcup$
	$q_5 \downarrow$				
X	0	X	$\sqcup$	$\sqcup$	$\sqcup$
		$q_{rej} \downarrow$			
X	$\sqcup$	X	$\sqcup$	$\sqcup$	$\sqcup$

(c) (*Graded for correctness*) Specify an example string w_3 of length 3 over Σ on which the

computation of this Turing machine is **never halts** or explain why there is no such example. A complete solution will include either (1) a precise and clear description of your example string and a precise and clear description of the looping (non-halting) computation of the Turing machine on this string or (2) a sufficiently general and correct argument why there is no such example, referring back to the relevant definitions.

Solution: Consider the string 100. Below is the looping computation of T on 100:

$q_0\downarrow$					
1	0	0	␣	␣	␣
	$q_3\downarrow$				
1	0	0	␣	␣	␣
$q_3\downarrow$					
1	0	0	␣	␣	␣
$q_3\downarrow$					
1	0	0	␣	␣	␣
...					

By convention, if the tape head is at the leftmost cell and transition function says it should move left, the tape head stays in the location. Therefore, given the self loop on q_3, the tape head will stay in the left-most cell and the computation will stay in q_3 forever. The computation cannot enter q_{acc} or q_{rej}, and it never halts.

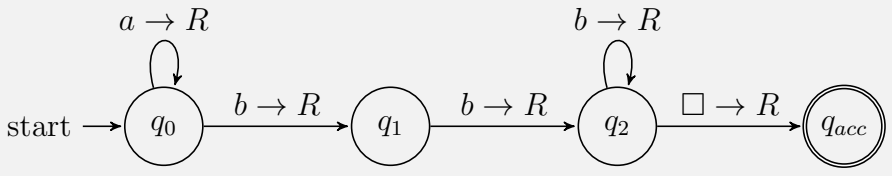
4. **Implementation-level descriptions of deciders and recognizers** (12 points): Consider the language

$$\{a^i b^j \mid i \geq 0, j > 1\}$$

over the alphabet $\{a, b\}$.

(a) (*Graded for correctness*) Give an example of a Turing machine that **decides** this language. A complete solution will include **both** a state diagram and an implementation-level description of this Turing machine, along with a brief explanation of why it recognizes this language, and why it is a decider.

Solution: Consider the input alphabet $\{a, b\}$ and tape alphabet $\{a, b, \square\}$:

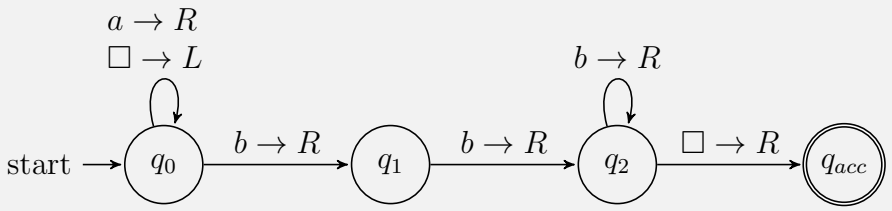


Implementation-level description: Scan the tape from left to right. If we see an a after b, reject. If we see less than or equal to one b before seeing a blank symbol, reject. Otherwise, accept when we see a blank symbol.

Explanation: The above Turing machine accepts all strings in the language $\{a^i b^j \mid i \geq 0, j > 1\}$ and rejects all strings not in the language, therefore it recognizes (and decides) this language. It is a decider because it does not loop on any input string.

(b) (*Graded for correctness*) Give an example of a Turing machine that **recognizes but does not decide** this language. A complete solution will include **both** a state diagram and an implementation-level description of this Turing machine, along with a brief explanation of why it recognizes this language, and why it is not a decider.

Solution: Consider the input alphabet $\{a, b\}$ and tape alphabet $\{a, b, \square\}$:



Implementation-level description: Starting at the left-most cell of the tape, if the symbol is blank, move the tape head to the left (which means staying at the left-most cell). Otherwise, scan the tape from left to right. If we see an a after b, reject. If we see less than or equal to one b before seeing a blank symbol, reject. Otherwise, accept when we see a blank symbol.

Explanation: The above Turing machine accepts all strings in the language $\{a^i b^j \mid i \geq 0, j > 1\}$, loops on ε, and rejects all other strings not in the language, therefore it recognizes this language. It is not a decider because it loops on ε.

5. **Classifying languages** (8 points): Our first example of a more complicated Turing machine was of a Turing machine that recognized the language $\{w\#w \mid w \in \{0,1\}^*\}$ (Figure 3.10 in the textbook), which we know is not context-free. Let's call that Turing machine M_0. The language

$$L = \{ww \mid w \in \{0,1\}^*\}$$

is also not context-free.

(a) (*Graded for correctness*) Choose an example string of length 2 in L that is in **not** in $\{w\#w \mid w \in \{0,1\}^*\}$ and describe the computation of the Turing machine M_0 on your example string. Include the contents of the tape, the state of the machine, and the location of the read/write head at each step in the computation.

Solution: Consider the string 11 which is in $L = \{ww \mid w \in \{0,1\}^*\}$ but not in $\{w\#w \mid w \in \{0,1\}^*\}$. The computation of M_0 on 11 is the following. We can see that the Turing machine rejects the string 11 because the computation enters q_{rej}.

$q_1 \downarrow$				
1	1	␣	␣	␣
	$q_3 \downarrow$			
x	1	␣	␣	␣
		$q_3 \downarrow$		
x	1	␣	␣	␣
			$q_{rej} \downarrow$	
x	1	␣	␣	␣

(b) (*Graded for completeness*) Explain why the Turing machine from the textbook and class that recognized $\{w\#w \mid w \in \{0,1\}^*\}$ does not recognize $\{ww \mid w \in \{0,1\}^*\}$. Use your example to explain why M_0 doesn't recognize L.

Solution: Since the Turing machine M_0 recognizes $\{w\#w \mid w \in \{0,1\}^*\}$, the set of all strings that are accepted by M_0 would contain exactly one # symbol. But all strings in $L = \{ww \mid w \in \{0,1\}^*\}$ do not contain #, which means that they will not be accepted by M_0. Take the example given in part (a), we can see that the computation of M_0 on the string enters q_{rej} after encountering the first blank symbol on the tape. This will happen for any string in L since M_0 do not see the expected # symbol. Therefore, all strings in L will be rejected by M_0, i.e. M_0 doesn't recognize L.

(c) (*Graded for completeness*) Explain how you would change M_0 to get a new Turing machine that does recognize L. Describe this new Turing machine using both an implementation-level definition and a state diagram of the Turing machine. You may use all our usual conventions for state diagrams of Turing machines (we do not include the node for the reject state $qrej$ and any missing transitions in the state diagram have value $(qrej, \square, R)$; $b \to R$ label means $b \to b, R$).

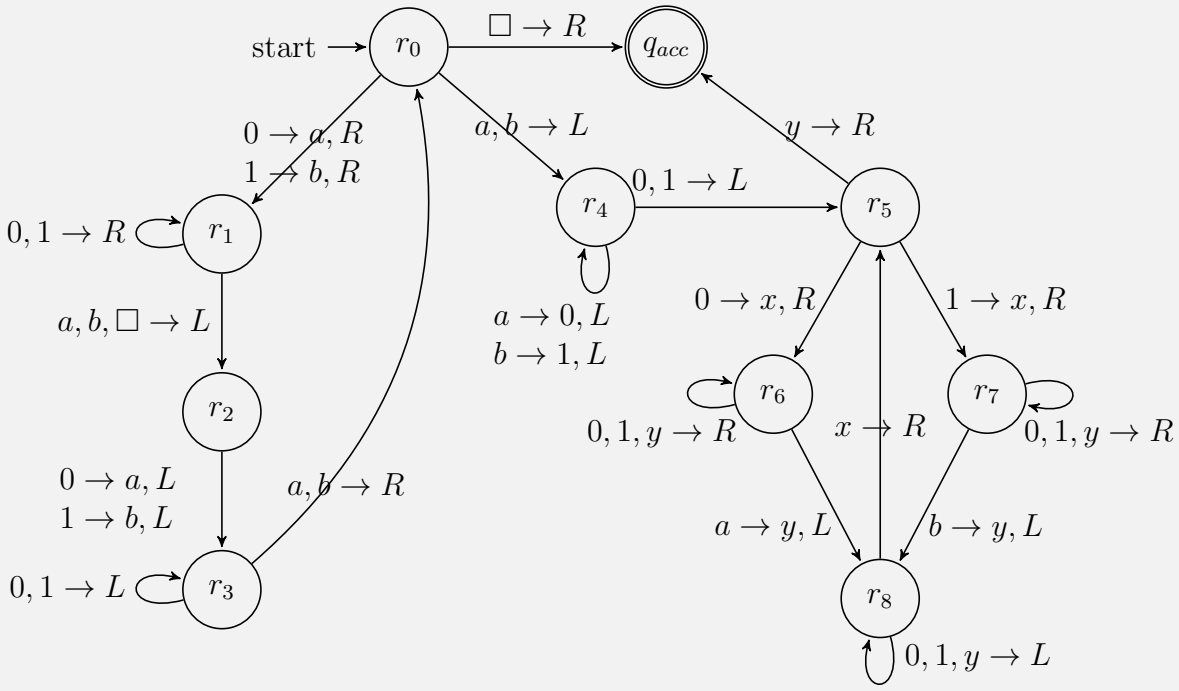
Solution: The new Turing machine need to first find the middle of the input string, and then compare the first half with the second half using the similar zig-zag strategy in M_0. The detail is as follows:

Implementation-level definition:
Zig-zag across the tape, mark the left-most symbol that is a 0 or 1 into a or b correspondingly, and mark the right-most symbol that is a 0 or 1 into a or b correspondingly. Repeat the above steps. If after marking a 0 or 1 on the left, there is no more 0 or 1 on the right to mark, reject, since the input is of odd length. Otherwise, after marking a 0 or 1 on the right and there is no more 0 or 1 on the left, the tape head is now pointing at the middle of the input. Scan left, convert the first half of the content back into 0's and 1's based on the marks (a becomes 0 and b becomes 1). The tape will now contain the original 0's and 1's representing the first

half of the input string, followed by a's and b's representing the second half. The tape head should be back to the left-most position at this point.

Then, zig-zag across the tape to corresponding positions on the first half and the second half to check whether a 0 in the first half matches with an a in the second half, and whether a 1 in the first half matches with a b in the second half. If they do not, reject. If all of the pairs match, accept.

State diagram:
Consider the input alphabet $\Sigma = \{0, 1\}$ and the tape alphabet $\Gamma = \{0, 1, a, b, x, y, \square\}$.



Here is a short explanation for the state diagram: Empty string will be accepted immediately. States r_0, r_1, r_2, r_3, r_4 process the input string into 0's and 1's in the first half and a's and b's in the second half. Note that computations on odd-length strings will enter q_{rej} after entering r_2 and not finding another 0 or 1 to mark. For even-length strings (that are not empty string), the computation will reach r_5, and r_5, r_6, r_7, r_8 corresponds to checking whether the first half and the second half match. Iff they match, the computation will enter q_{acc}.