Homework 2 Solutions

1. In the following answers, I assume an understanding of proof by contradiction using the closure properties of regular langauges. If this is an incorrect assumption, please meet with the instructor of your section.

   (a) Assume by way of contradiction that L is regular. Define $L_1 = a^*b^*a^*$, which is regular (can you think of a three-state DFSA that accepts it?). Now, define $L_2 = L \cap L_1$. Since L_1 is regular and L is assumed to be regular, L_2 must be regular, since regular languages are closed under intersection. Now, $L_2 = a^k b^j a^k$. Since L_2 is regular, by the pumping theorem there must exist a number $n > 0$ such that any string $w \in L_2$ longer than n symbols can be written $w = xyz$ such that:

   - $|xy| < n$
   - $y \neq e$
   - $xy^iz \in L_2$ for any $i \geq 0$

   Let's take some string w in L_2 with $k > n$. Then, $w = a^k b^j a^k$ can be broken into x, y and z parts such that x and y consist only of a's (why?). Say that y has length $|y|$. We know by the pumping lemma that xz is also in L_2. But xz is $k - |y|$ a's followed by j b's, followed by k a's. This string is not in L_2. So we've arrived at a contradiction, and thus our original assumption, that L was regular, must have been false.

   Though we don't show it, a similar proof can be used for each of the remaining parts. Almost everyone got this problem wrong, and it's suggested that you re-read L&P to make sure you understand how to apply the pumping lemma.

2. We can show that the language specified is regular by constructing it, using closure properties, from other regular languages. First, a DFSA recognizing the language, L_1, strings of numbers divisible by 3, is shown on page 87 of *Lewis and Papadimitriou*. Second, we can specify a regular expression that accepts strings that *have* double numbers in them as follows: $L_2 = \Sigma^*(00 \mid 11 \mid 22 \mid 33 \mid 44 \mid 55 \mid 66 \mid 77 \mid 88 \mid 99 \mid 00)\Sigma^*$. Third, because regular languages are closed under complement, we know that $L_3 = \bar{L_2}$, which accepts strings that do *not have* double numbers in them, is regular. Fourth, and finally, the language we are trying to construct, L, is simply $L = L_1 \cap L_2$, which is therefore regular.

3. (a) We can break this into two cases, one where $m < n$, and one where $m > n$. In both cases, the string is made up of one substring, E containing an equal number of a's and b's, and another containing $m - n$ a's (A) or $n - m$ b's (B). Written out, this becomes:

$$\begin{aligned} S &\longrightarrow AE \mid EB \\ A &\longrightarrow aA \mid a \\ B &\longrightarrow Bb \mid b \\ E &\longrightarrow aEb \mid e \end{aligned}$$

(b) This question also required growth regulation on a and b; a had to grow at least as b, but no more than twice as fast as b. Similar to above, the intuitive rule to use is $S \longrightarrow aSb \mid aaSb \mid e$. The tricky part, however, is to look at the base case. The shortest string that satisfies the $m < n < 2m$ requirement is *aabbb*; anything shorter is either $n = m$ or $m = 2n$. So, we start with the base case and grow as we would expect:

$$\begin{aligned} S &\longrightarrow aaS_1bbb \\ S_1 &\longrightarrow aS_1b \mid aaS_1b \mid e \end{aligned}$$

(c) In this case, the requirement moves from left to right, so our construction rules should as well. In particular, the "prefix" must have as many a's as b's. Well, then, make sure that everything left of a string, S, does just that:

$$S \longrightarrow aS \mid abS \mid e$$

4. For one example, there are two distinct parse trees for (()):

$$SS \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())$$
$$SS \Rightarrow SS \Rightarrow S(S) \Rightarrow S((S)) \Rightarrow S(()) \Rightarrow (())$$

To disambiguate this CFG, we start by observing that the beginning of any string, S, must be an open parentheses: $S \longrightarrow (R$. For the right side of that string, R, we can either close the string, $R \longrightarrow)$, close it and open a new one, $R \longrightarrow)(R$, or leave it open and insert a valid string inside of our current one, $R \longrightarrow (S)R$. By dealing with th left and right side of a valid string separately, we ensure that the parse tree must move from left to right. Putting it all together,

$$S \longrightarrow S(S) \mid e$$

Note that a derivation is not the same as a parse tree; a string that has more than one derivation but only one parse tree does *not* make a CFG ambiguous.

5. In order to solve this problem, you must use an NPDA (*remember that an NPDA is more powerful than a DPDA, unlike the same relationship for FSAs or Turing Machines*). The key is to recognize that there is no way for a DPDA to recognize when it has come to the middle of a string, while an NPDA can try all of the possibilities by cloning.

We can build this PDA in four stages. This PDA is so simple that you might describe it entirely through it's transition function. First, we consider the "left half of the palindrome" state, s_0, that allows us to push characters into the stack:

$$((s_0, a, e), (s_0, a))$$
$$((s_0, b, e), (s_0, b))$$

Second, we consider the center of an odd palindrome that requires us to discard the center character and move to the right side of the palindrome:

$$((s_0, a, e), (s_1, e))$$
$$((s_0, b, e), (s_1, e))$$

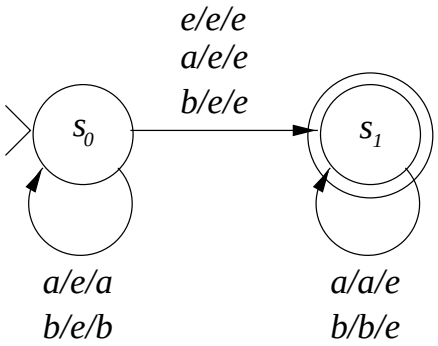
Third, we consider the center of an even palindrome that requires us to move from the left side of the palindrome to the right side of the palindrome without using any characters in w. This transition will also take care of the empty string, which is a degenerate palendrome of sorts:

$$((s_0, e, e), (s_1, e))$$

Fourth, we consider the "right side of the palindrome" state, s_1, that allows us to read characters only so long as they match what we pop from the stack:

$$((s_1, a, a), (s_1, e))$$
$$((s_1, b, b), (s_1, e))$$

Of course, the only accept state is $F = s_1$. We can draw this NPDA just as easily (*input/pop/push*):



A few notes about this problem that should clarify some misconceptions:

- We do not have to check for an empty stack at the end, since the accept condition for a PDA requires both being in an accept state and having an empty stack.
- We can tell that this is an NPDA (rather than a DPDA) because there are multiple transitions from s_0 for both of the input characters, a and b.
- This problem is similar to examples 3.3.1 and 3.3.2 in *Lewis and Papadimitriou.*

6. One of the more fortuitous mistakes in *L*ewis and Papadimitriou is in example 4.1.8. Rather than implement a "left-shifting machine" as they say they are, they create a palindrome maker that performs $f(w) = ww^R$. By using their machine and removing the initial w, we

have our flipping machine. In the drawing below, the top part of the diagram is the machine, while the bottom scans to the left and erases the original string.