Question 1: Convert PDA to CFG

Part a:
What language does this recognize? Hint: it is a language that was presented in the class slides.

Part b:
Give a grammar for this language. Note that the grammar that you will generate with Lemma 2.27 will be similar to your grammar, but might not be as concise.

Part c:
To convert a PDA to a CFG, lemma 2.27 requires it be first converted into a special form. State what that form is. Is the PDA above in this form?

Part d:
In Lemma 2.27, you need to add a set of rules corresponding to Figure 2.28 (2nd part). How many rules will this create? Explain your answer.

Write the rules that will actually be useful. Use $S_{ab}$ to format $S_{ab}$.

Part e:
Write all of the rules that correspond to the 3rd part (the epsilon rules).
Which ones will be useful?

Part f:
Write all of the rules that correspond to Figure 2.29 (first part). For each rule, explain why you were able to add it.

Part g:
What is the start variable?

Pumping Lemma for Context Free Languages

For each of these questions, I will assume that you are starting off your proof with the following, so you don’t have to write it:

Assume that $L$ is CF.
Therefore, there exists a $p$ such that for any string $s \in B$ where $|s| \geq p$, it can be written as $uvxyz$ where $|xy| \leq p$ and $|y| \geq 1$.

Let $s$ be ...
Question 2: Problem 2.30 Part a

Question 3: Problem 2.31

Question 4: Problem 2.32

Show Languages are CF by constructing a CFG or PDA

Question 5: Problem 2.22

There are 3 ways in which $x$ and $y$ can differ. $x$ can be shorter than $y$, $x$ can be longer than $y$, and $x$ and $y$ can differ in some position (may or may not be the same length). Part a and b are fairly straight-forward. But, c is the tricky part.

Part a.
Give a PDA in which $x$ is shorter than $y$.

Part b.
Give a PDA in which $x$ is longer than $y$.

Part c.
The third case is tricky. So, do the following easier version first. Construct a PDA that recognizes $D = \{ x\#y | x, y \in \{0, 1\}^* \text{ and } |x| = |y| \text{ and } x \neq y^R \}$.

In other words, give a PDA that can generate $x\#y$ where (a) $x$ and $y$ are of the same length and (b) $x$ and $y$ differ in at least one position: the $i$th position of $x$ from the beginning of $x$, and the $i$th position of $y$ from the end of $y$.

Part d.

We will now do the third case. Normally, when you think of a PDA, you think in terms of how one part of a string relates to the reversal of an earlier part of the string. This is why part c was included. However, to do the third case, part c is not relevant at all. That question was included so that you could be told that thinking of this problem in that way is not helpful.

For the third case: let $E = \{ x\#y | x, y \in \{0, 1\}^* \text{ and where } x = x_1...x_k \text{ and } y = y_1...y_j \text{ and } x_i, y_i \in \{0, 1\} \text{ and } x_i \neq y_i \text{ for some } i \leq k \text{ and } i \leq j \}$. Make a PDA that recognizes $E$.

In other words, make a PDA that makes sure that there is some $i$ such that the $i$th character from the beginning of $x$ is a 1, then the $i$th character from the beginning of $y$ is a 0. Or vice versa, where the $i$th character of $x$ is a 0, and the $i$th character of $y$ is a 1. The PDA does not need to make sure that $x$ and $y$ are of the same length.

Part e.
Now explain how to construct the overall PDA from your PDAs for parts a, b, and d. You do not have to give the actual construction.

Explain why you need to include parts a and b, especially as part d does not require $x$ and $y$ to have the same length. In other words, give a string $s_a$ that accepted by your PDA from part a and another string $s_b$ accepted by your PDA from part b, and explain why neither is accepted by your PDA from part d.

Question 6: Problem 2.23

Note that $ww$ is not regular. So, anything based on that will not work.
In fact, my grammar (or PDA) does not know where the center of the string is. In addition to giving a PDA or a grammar, give a brief explanation of why your PDA or grammar does generate the language.

**Question 7: Problem 2.21**

**Part a.**

Give a PDA that recognizes the language with the same number of *a*’s and *b*’s. Carefully think about what the stack should keep track of. Think in terms of the minimum amount of information you would keep track of as you are reading an input one character at a time to see if it is in the language. In fact, think of how you would process the following input:

```
aabaabaaabbbbbbaa
```

For specifying your PDA, as a shortcut, you can allow your PDA to push multiple characters onto the stack in one operation. This is useful if you want your PDA to see what is on the top of the stack, and then keep that character on the stack while pushing another character on as well. For example, *a, $ → $a* means if the next input is an *a* and $ is on top of the stack, keep $ on top of stack, and push *a* on top of that.

Your PDA only needs 3 states, with the first and last states just being used to push and pop a ‘$’ onto the stack so that we can test for an empty stack while reading $\epsilon$. Other than the transition to the accept state, your PDA should be deterministic, for any input and any top-of-stack, there should just be one transition for it.

**Explain** how you will use the stack. What stack characters are you using, and what does each mean? What types of strings are allowed on the stack?

Give a state diagram for your PDA.

**Part b.**

Give a CFG that recognizes the language with the same number of *a*’s and *b*’s. Rather than just try to make up a set of rules, think in terms of the proof of lemma 2.27. When you read an ‘a’, you eventually have to read its matching ‘b’. The string that is in between the matching *a* and *b*, will also be in the language.

**Part c.**

Make a PDA that recognizes the language with twice as many *a*’s as *b*’s. Clearly explain the different stack symbols you are using, and what they mean. Also, explain what sequences of stack symbols are allowed by your PDA.

Again, make it work deterministically.

**Part d.**

Let *L* be the language with twice as many ‘a’s as ‘b’s.

Give a CFG for *L*. This grammar is actually just a slight alteration of the grammar for part b.

**Bonus Part e.**

Prove that your grammar is correct. For this, you need to prove that (1) any string generated by your grammar is in *L* and (2) any string in *L* can be generated by your grammar. The first part is not too difficult.

For the second part, here are some hints.

Let’s *s = w_1w_2...w_n*. Define *c(s)* as the count of the number of ‘a’ minus two times the number of ‘b’. So, *L = {s | c(s) = 0}*. Do a proof by induction based on the length of the string. The basis (length 0) is pretty easy. For the
induction, think of breaking $s$ into two nonempty substrings $s_1, s_2$ such that $s = s_1s_2$. Think of some cases for $c(s_1)$, and prove that for each case, the resulting string is in $L$. 