CS 3100, Fall 2018, Assignment 4 – 150 pts total

1 Asg-4 Questions

1. **25 points:** (Alex)

   (a) **8 points:** Write down a context-free grammar for the language L which is exactly all those strings over $\{a, b\}^*$ in which there are twice as many `b`'s as `a`'s.

```
S -> '' | X | Y | Z
X -> Sabb | aSbb | abSb | abbS
Y -> Sbba | bSba | bbSa | bbaS
Z -> Sbab | bSab | baSb | babS
```

   (b) **5 points:** Argue that this context-free grammar is consistent. Use 3-4 sentences that indicate how the recursive "templated" nature of the grammar allows you to argue consistency.

   This grammar will not include anything outside the described language. This is because it is impossible to assemble a string that does not have twice as many b's as a's from this grammar. Every possible production adds two b's for every one a.

   (c) Argue completeness as follows You may draw a hill/valley plot in which you must plot a hypothetical (example) string beginning with an `a`. For every `a`, imagine rising two steps, and for every `b`, fall one step. (only **two** of the cases are requested):

   - **6 points:** **(Case 1 of 2 requested):** Consider the case of no zero-crossings with the example string beginning with an `a`.
     Draw a parse tree and argue completeness. Then write 2-3 sentences against this drawing.

   Parse Tree: "aabbbbabb"

```
S
|
X
|
Sabb
|
X
|
aSbb
```

```
|
X
|
abbS
   |
   S
   |
   ''
```

This string is obviously included in the grammar. If we consider a string that is 3 character longer, one a and two b's, this string is also in the language, and in the grammar. Whether or not they introduce a zero crossing, these letters will take 2 steps up and then 2 steps down, returning to the x axis of the hill valley plot.

- **6 points:** **(Case 2 of 2 requested):** Now consider exactly one zero-crossing, and write the proof of completeness. Again the example string must start with an `a`, then zero-cross and travel below the x-axis, and return to the x-axis. **Again draw a parse tree and argue completeness.** Then write 2-3 sentences against this drawing.

Parse Tree: "abbbba"

```
S
|
X
|
abbS
   |
   Y
   |
   Sbba
   |
   S
   |
   ''
```

Again here we can see that this string is included in the language and the CFG. WE can prove this again by imagining a string 3 character shorter that we know is in the CFG. These 3 character must be one a and two b's, which will move the plot away from the x axis, then back to it.

2. **25 points:** (Paul)

   (a) **12 points:** Develop all parse trees for the sentence $1+\sim 2*3$ using **CfgExp1** from Section 11.5 of our book, and **CfgExp2** from Section 11.5.1 of our book. Write down all possible answers produced by employing these grammars.
   You must draw the parse tree(s) for each case, and write 2-3 sentences explaining why these are the parse trees for the given sentence.

Trees with CfgExp1:

```
    E
    |
    E + E
    |    \
    1    E * E
         |   |
         ~E  3
         |
         2

    E
    |
    E * E
   /     \
E + E     3
|   |
1   ~E
    |
    2
```

With CfgExp1 you can start with multiplication, addition, or $\sim$. However, starting with $\sim$ here cannot produce the desired string, so these are the only possibilities.

Trees with CfgExp2:

```
    E
    |
    E + T
    |    \
    T     T * F
    |     |    \
    F     F     3
    |     |
    1     ~F
          |
          2
```

CfgExp2 in unambiguous, so there is only one possible parse tree.

(b) **6 points:** Develop a CFG for the language L_{wwR}:

$$L_{wwR} = \{ \ ww^R \ : \ w \in \{0,1\}^* \ \}$$

You must present the CFG and write about 2 sentences describing it.

```
S -> 1 S 1 | 0 S 0 | ''
```

This CFG captures all strings which are palindromes over $\{0,1\}*$. It works from the inside out matching the end of the original string to the beginning of the reversed string.

(c) **7 points:** Parse `aaabbbccc` using **CFGabORbc** from Section 11.6 of our book in two ways. Think of how many ways you can parse `aaabbbccc` using this grammar.[1]
You must either present two different derivation sequences or draw two different parse-trees. Then you must write two sentences explaining your work.

Parse trees:

```
        S
        |
     Mab Cs
    /     \
a Mab b    c Cs
   |         |
a Mab b    c Cs
   |         |
a Mab b    c Cs
   |         |
  ''       ''


      S
      |
    As Mbc
   /     \
  a As    b Mbc c
   |         |
  a As    b Mbc c
   |         |
```

[1]Comment: This question at least helps see why L_{abORbc} is inherently ambiguous. The *proof* that L_{abORbc} is inherently ambiguous is much more elaborate. We must not merely consider **CFGabORbc**, but *all possible* CFGs must be considered in such a proof, and all must be shown to result in ambiguity.

```
a As    b Mbc c
 |         |
 ''        ''
```

This CFG has two possible choices at the top most level, both of which capture this string. Therefore there are two parse trees.

3. **25 points:** (Paridhi)

(3.5 points per CF part; 6 points per non-CF part)

For the languages below that are CFLs, write down the CFG. For languages that are not CFLs, write a Pumping Lemma proof of sufficient clarity. This consists of writing the general strategy to pick a w, and list all the cases ("splits") to be considered. Then show that by pumping, we go outside the language for one of the cases (if there is only one case). If there are multiple cases, discuss them all and and present the detailed proof for at least two of the cases. Note: $\Sigma = \{0,1\}$

(a) $L_{P0} = \{w \ : \ w \in \Sigma^*\}$

CFG:
```
S -> 0 S | 1 S | ''
```

(b) $L_{P2} = \{waw^R \ : \ a \in (\{\varepsilon\} \cup \Sigma), \ w \in \Sigma^*\}$

CFG:
```
S -> 1 S 1 | 0 S 0 | A
A -> 0 | 1 | ''
```

(c) $L_{ww} = \{ww \ : \ w \in \Sigma^*\}$

This is not a CFL

i. We can chose w as $0^p1^p0^p1^p$, where p is the pumping length associated with a CFG for this language.
ii. The v and y blocks could either fall within the first 0^p1^p block, the middle 1^p0^p block, or the last 0^p1^p block. Within each of those cases v and y could fall entirely within a 0^p or 1^p block.
iii. In any of these cases, pumping up or down will result in a string that is not one string duplicated.
iv. In the first case, where v and y fall within the first 0^p1^p block, pumping up or down will cause the first w to not equal the second w
v. Similarly, in the case where v and y fall within the last 0^p1^p block, pumping up or down will mean the second w does not equal the first one.

(d) $L_{eq010} = \{0^n1^n0^n \ : \ n \geq 0\}$

This is not a CFL

i. Chose $w = 0^p1^p0^p$, such that p is the pumping length for come CFG
ii. Here v and y can either fall entirely in a 0^p or 1^p block, or spread across two of those. In any case, pumping up or down will result in a string where there are not the same amount of 0's, 1's, and 0's.
iii. In the case where v and y fall entirely within the first 0^p, pumping up or down will mean that the first block is not the same length as the second or third.
iv. Similarly in the case where v and y fall entirely in the middle block of 1's, pumping up or down will result in a string that does not have the correct amount of 1's.

(e) $L_{eq012} = \{0^n1^n2^n \; : \; n \geq 0\}$

This is not a CFL

i. Chose $w = 0^p1^p2^p$, such that p is the pumping length for come CFG
ii. Here v and y can either fall entirely in the 0^p block, 1^p block, or 2^p block. It could also be spread across two adjacent blocks. In any case, pumping up or down will result in a string where there are not the same amount of 0's, 1's, and 2's.
iii. In the case where v and y fall entirely within the first 0^p, pumping up or down will mean that the first block is not the same length as the second or third.
iv. Similarly in the case where v and y fall entirely in the middle block of 1's, pumping up or down will result in a string that does not have the correct amount of 1's.

A writeup "CFL Pumping Lemma Without the Pain" is here: `https://v1.overleaf.com/read/bwdqfjkgsppc`, and it might also teach you how to write these PL proofs.

4. **25 points** (Arnab) Exercise 11.10.1, CFG design, Problem 3: *Develop a CFG for the language of strings over $\Sigma = \{0, 1\}$ where the number of 1's is strictly greater than the number of 0's.* There are hints provided in the book on how to approach this problem. You may consider constructing "hill/valley" plots for such strings. It will demonstrate that there could be segments of the plot that represent equal 0's and 1's, and other segments that steadily climb up. It might be a good idea to introduce separate nonterminals for these parts. Then, when you assemble, think of the overall plot such that

   - It can start with an "equals" segment or a "rising" segment
   - It can have more "equals" and "rising" interspersed.
   - It can end in an optional "equal segment"

   Questions

   - (8 of the 25 points) Designing the CFG and describing your design in 4-5 clear sentences.

```
S -> A R | R A | R
A -> 0 A 1 | 1 A 0 | AA | R | ''
R -> 1
```

In this CFG the A production represents any "equals" segments, and R represents the "rising" segments. This CFG could likely be simplified, but I think this representation is easier to understand. The A production will generate all strings with equal 0's and 1's. R can simply add a 1 anywhere in the string, as many times as desired. With this combination, all strings with strictly more 1's than 0's can be generated.

   - (3 of the 25 points) Argue in your own words why your solution is complete and consistent(Formal Proof is not required)

This solution is consistent because it is impossible to generate a string that does not have more 1's than 0's. The R production guarantees this by always having at least 1 one outside of the "equal" portions. This solution is consistent because any possible string over {0, 1} with more 1's than 0's can be generated. The A production will generate any string over {0, 1} with an equal number of 0's and 1's, and allow the R production to insert as many 1's as desired anywhere in that string.

Show how your CFG handles the following input strings (spaces added for clarity) by drawing parse-trees for these inputs.

- (1 of the 25 points) `1`

```
            S
            |
            R
            |
            1
```

- (3 of the 25 points) `01101`

```
            S
            |
            AA
          /   \
      0 A 1    0 A 1
        |        |
        R        ''
        |
        1
```

- (4 of the 25 points) `0110011`

```
            S
            |
            AA
          /    \
        AA     0 A 1
      /   \        \
0 A 1     1 A 0     R
  |         |       |
  ''       ''       1
```

- (6 of the 25 points) `01110110`

```
            S
            |
            AA
        /      \
     0 A 1      AA
       |       /     \
      AA     0 A 1    1 A 0
```

```
          /  \       |         |
         R    R      ''        ''
         |    |
         1    1
```

5. **25 points** (Harshitha) Design a PDA using the direct approach (without converting from a CFG) for the set of strings over $\{a, b, c\}$ where the number of a's equals the number of b's plus the number of c's. That is, $\#_a = \#_b + \#_c$.

$$\{w \in \{a, b, c\}^* \ : \ \#_a = \#_b + \#_c\}$$

Your answer must be a Jove file called `J5_UNID.ipynb` with this problem clearly identified in it, and your solution + your tests clearly shown. You must test your created PDA sufficiently well. All code cells must be run before you submit. It must also pass the following tests (spaces added for clarity):

- `''`
- `b a a c`
- `a b a c a b b c b c c c c b a a a a a a a a`
- `a a a a b a a c a b b c b c c c c b a a a a`

6. **25 points:** (Maryam)

(a) **(10 points)** Design a context-free grammar for the language of strings over $\{a, b, c\}$ where the number of a's equals twice the number of b's plus the number of c's.

$$\{w \in \{a, b, c\}^* \ : \ \#_a = 2\#_b + \#_c\}$$

CFG:

```
S -> X | Y | Z | SS | ''
X -> a S c | c S a
Y -> aa S b | ab S a | ba S a
Z -> b S aa | a S ab | a S ba
```

(b) **(15 points)** Convert this CFG into a PDA using the direct CFG to PDA conversion method (done by hand). Then enter the obtained PDA in Jove's syntax into `J5_UNID.ipynb`. You must clearly identify this problem in this Jove file, and your solution + your tests must be clearly shown. You must test your created PDA sufficiently well. All code cells must be run before you submit. It must also pass these tests:

- `''`
- `a a b b a a`
- `a c a c a c a c`
- `a b a b a a c a c a c a`
- `a c a b a c a b a c a b a a a`

Also make sure that the PDA rejects all the tests of Question 5 except for epsilon.

Change log since Asg4 was issued:

1. Changed Q3 to update the Σ. Also updated the pumping lemma proof requirements for full credit.
2. Changed wording of Q2 Part C to clarify that you are only required to parse `aaabbbccc` in two ways.
3. Added more hints for Q4 CFG design. Changed wording for Q4 consistency and completeness requirement. Points granularity added for each subpart of Q4.
4. Added the correct tests for Question 6.