Aughdon Breslin and Isabella Cruz
"I pledge my honor that I have abided by the Stevens Honor System."

**Problem 1**
(5 points) If $G$ is a CFG in Chomsky Normal Form, show that a string of terminal symbols of length $n \geq 1$ is generated by the application of exactly $2n - 1$ rules of $G$.

In order for a language to be in Chomsky Normal form its rules must be of the form A→ BC or A → a where A,B, and C are variables and a is a terminal symbol.

<u>If: n = 1, 2(1) - 1 = 1</u>
To get a string of length 1, you only need to apply one rule in A → a form.
EX: A → a yields the string "a", which has a length of 1

<u>If: n = 2, 2(2) - 1 = 3</u>
In order to get a string of length two, you would apply one rule in A → BC form, then apply two rules in A → a form.
EX: A → BC, B → b, C → c yields the string "bc", which has a length of 2

<u>If: n= 3, 2(3) - 1 = 5</u>
In order to get a string of length three, you would apply two rules in A → BC form and three in A → a form.
EX: A → BC, B → CD, C → c, D → d yields the string "cdc", which has a length of 3

The pattern shown is that in order to get a string of length "n", you must apply "n-1" rules in A → BC form and "n" rules in A → a form. The pattern works for any n because each A -> BC formatted rule essentially provides one more space for a direct translation, where C -> c acts as the translation. Each additional space is created using 2 additional rules, with the exception of a length 1 string only requiring the translation, hence the 2n-1 formula.

Therefore, a string of terminals of length n for all n ≥ 1 is generated by applying exactly (n-1) + n rules = 2n - 1 rules.

**Problem 2**
(10 points) Let $G = (V, \Sigma, R, S)$ be a CFG where $V = \{S, T,U\}$, $\Sigma = \{0, \#\}$, and $R$ is the set of rules:

$$S \to TT \mid U$$
$$T \to 0T \mid T0 \mid \#$$
$$U \to 0U00 \mid \#$$

Describe the language $L(G)$ in English and prove that it is not regular.

A: The language can follow one of two rulesets.

In the case of TT: The language consists of a series of zeroes with exactly 2 hash symbols anywhere inside the string. The language can be represented by $\{0^i\#0^j\#0^k;\ i, j, k >= 0\}$.

In the case of U: The language consists of some multiple, n, of 3 zeroes, where n >=0, with a hash symbol after exactly ⅓ of the zeroes. The hash will also be there if n = 0. The language can be represented by $\{0^i\#0^{2i};\ i>= 0\}$.

Overall, L(G) is either a language that consists of a series of zeroes of size n>=0 with exactly 2 hash symbols positioned anywhere within the string, or a multiple, n >=0, of 3 zeroes with exactly 1 hash symbol positioned after exactly ⅓ of the zeroes. If n = 0 in the second case, then the string would just consist of a hash symbol. The language can be represented by $\{0^i\#0^j\#0^k;\ i, j, k >= 0\} \cup \{0^i\#0^{2i};\ i>= 0\}$.

<u>Finding regularity of TT, U</u>
In the case of TT = $\{0^i\#0^j\#0^k;\ i, j, k >= 0\}$:

TT can be represented as 0*#0*#0*. Therefore TT is a regular language.

In the case of U = $\{0^i\#0^{2i};\ i>= 0\}$:

Assume U is regular.
By the Pumping Lemma, U has a pumping length, p.
Let $s = 0^p\#0^{2p} \in A$
$s = xyz$, $|y| > 0$, $|xy| <= p$
y must consist of some number of 0s, $y = 0^k$ where $1 <= k <= p$.
$xz = 0^{p-k}\#0^{2p}$.
For 1 <= k <= p, xz does not follow the ruleset of the language since there will no longer be both i 0s on the left of the # and 2i 0s on the right. This violates the pumping lemma. Therefore U is not a regular language.

Finding the regularity of L(G)

We assume L(G) is regular. The language of TT is mutually disjoint from the language of U since TT must contain exactly 2 hash symbols while U must contain exactly 1.

L(G) = TT ∪ U --> L(G) - TT = U.

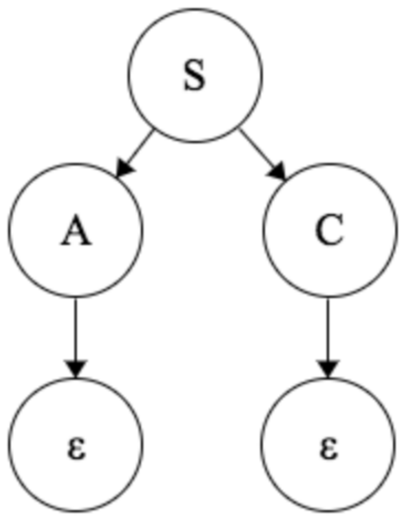
Since we assume L(G) to be regular, and we've shown TT is regular, then L(G) - TT is regular under closure of difference. However, the left hand side of the equation is equivalent to the non-regular language U, which is a contradiction. Therefore, our assumption must be wrong, and L(G) must be non-regular.

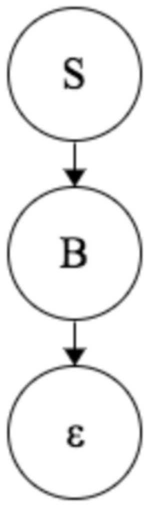
**Problem 3**

(10 points) Give a CFG for { $a^ib^ic^kd^k$ : $i, k \geq 0$} ∪ { $a^ib^kc^kd^i$ : $i, k \geq 0$}. Is your grammar ambiguous?

| | |
|---|---|
| S -> AC \| B | //go into the first or second language ruleset |
| A -> ε \| aAb | //as many a's followed by equal b's |
| C -> ε \| cCd | //as many c's followed by equal d's |
| B -> ε \| aBd \| aDd | //as many a's coupled with ending d's |
| D -> ε \| bDc | //as many b's followed by equal c's |

Yes. ε can be rep'd in multiple ways, therefore the grammar is ambiguous.





**Problem 4**

(15 points) Let $L_{add} = \{ a^i b^{i+j} c^j : i, j \geq 0\}$ and $L_{mult} = \{ a^i b^{ij} c^j : i, j \geq 0\}$. For each language, either give a CFG for it, or prove that it is not a CFL.

$L_{add}$:

S -> AB

A -> ε | aAb    //series, length i, of a's followed by same number of b's

B -> ε | bBc    //series, length j, of b's followed by same number of c's

$L_{mult}$:

$L_{mult}$ is not a context-free language.

Proof

Assume $L_{mult}$ is a context-free language.

By the pumping lemma, there exists a pumping length p for $L_{mult}$

Let's say $S = a^p b^{p*p} c^p$ and $S \in L_{mult}$ → $\underbrace{\text{| a a ... a a |}}_{p} \underbrace{\text{| b b ... b b |}}_{p*p} \underbrace{\text{| c c ... c c |}}_{p}$

$|s| \geq p$, $\exists$ i: $uv^i xy^i z$ where $|vy| > 0$ and $|vxy| \leq p$

Where can the segment vxy lie within the string?

Possible Windows

Note: _ represents a possible window, |'s are just for organization, not actually part of the string

v and y contain different symbols of $\Sigma_{mult}$ → | a a ... a a || b b ... ... b b || c c ... c c | :

When v and y contain different symbols, if the string is pumped up (uvvxyyz, uvvvxyyyz, etc.) the symbols a & b or b & c will be out of order. This breaks the ruleset of $L_{mult}$ , so the string cannot be a member of $L_{mult}$.

v and y contain only a's → | a a ... a a | | b b ... b b | | c c ... c c | :

If v and y contain only a's, vxy must consist of $a^k$ where $1 \leq k \leq p$. If the string is pumped up ($uv^2xy^2z$ or uyyxyyz), the string will become $a^{p+k} b^{p*p} c^p$. This violates the rule that the number of b's = i * j because (p+k) * p does not equal p*p.

v and y contain only b's → | a a … a a | | b b … b b | | c c … c c | :

If v and y contain only b's, vxy must consist of $b^k$ where $1 \le k \le p$. If the string is pumped down ($uv^0xy^0z$ or uxz), the string will become $a^p b^{p*p-k} c^p$. This violates the rule that the number of b's = i * j because p * p does not equal p*p-k.

v and y contain only c's → | a a … a a | | b b … b b | | c c … c c | :

If v and y contain only c's, vxy must consist of $c^k$ where $1 \le k \le p$. If the string is pumped up ($uv^2xy^2z$ or uyyxyyz), the string will become $a^p b^{p*p} c^{p+k}$. This violates the rule that the number of b's = i * j because p * (p+k) cannot equal p*p.

Therefore, since there is no window in the string where vxy can be placed and pumped while continuing to follow the ruleset, the pumping lemma is violated. Thus, $L_{mult}$ is not a context free language.

**Optional Problem 5**
(10 points) Let Σ = {*a*, *b*}. Give a CFG to generate all and only strings which contain twice as many *a's* as *b's*. Give a proof that your grammar is correct.

Note: 0 x 2 = 0, so ε is included in the language

S -> ε | Saab | aSab | aaSb | aabS
| Saba | aSba | abSa | abaS
| Sbaa | bSaa | baSa | baaS

Proof
PROVE THAT NO MATTER WHAT RULE OR COMBO OF RULES YOU USE FOR S, YOU GET $b^i$ AND $(aa)^i$ WHERE i ≥ 0

ε :
ε follows the rules of a language that has twice as many a's as there are b's because if there are 0 b's, 2 x 0 = 0, so there should be 0 a's. Thus, since there are 0 a's and 0 b's, the string would be empty.

---------------------------------------- aab Segment ----------------------------------------

Saab:
"Saab" follows the rules of a language that has twice as many a's as there are b's because every time you use this rule, you add 1 b preceded by 2 a's. If you were to pump this rule alone you would produce the string $(aab)^i$ where i ≥ 0 which will always have twice as many a's than b's.

aSab:
"aSab" follows the rules of a language that has twice as many a's as there are b's because every time you use this rule, you add 1 "a" to the front portion of the string and the string "ab" to the back portion. Essentially, you are adding two a's for every b in the string. If you were to pump this rule alone you would produce the string $a^i(ab)^i$ where i ≥ 0, which will always have twice as many a's than b's.

aaSb:
"aaSb" follows the rules of a language that has twice as many a's as there are b's because every time you use this rule, you add 2 a's to the front portion of

the string and 1 b to the back portion. Essentially, you are adding two a’s for every b in the string. If you were to pump this rule alone you would produce the string $(aa)^i b^i$ where $i \geq 0$, which will always have twice as many a’s than b’s.

---------------------------------------- aba Segment ----------------------------------------

<u>Saba:</u>
“Saba” follows the rules of a language that has twice as many a’s as there are b’s because every time you use this rule, you add the string “aba” to the back of the string, which adds 2 a’s for every b. If you were to pump this rule alone you would produce the string $(aba)^i$ where $i \geq 0$, which will always have twice as many a’s than b’s.

<u>aSba:</u>
“aSba” follows the rules of a language that has twice as many a’s as there are b’s because every time you use this rule, you add one a to the front portion of the string and the string “ba” to the back portion. Essentially, you are adding two a’s for every b in the string. If you were to pump this rule alone you would produce the string $a^i(ba)^i$ where $i \geq 0$, which will always have twice as many a’s than b’s.

<u>abaS:</u>
“abaS” follows the rules of a language that has twice as many a’s as there are b’s because every time you use this rule, you add the string “aba” to the end of the string, which adds 2 a’s for every b. If you were to pump this rule alone you would produce the string $(aba)^i$ where $i \geq 0$, which will always have twice as many a’s than b’s.

<u>Sbaa:</u>
“Sbaa” follows the rules of a language that has twice as many a’s as there are b’s because every time you use this rule, you add 1 b followed by 2 a’s. If you were to pump this rule alone you would produce the string $(baa)^i$ where $i \geq 0$ which will always have twice as many a’s than b’s.

---------------------------------------- baa Segment ----------------------------------------

<u>bSaa:</u>

“bSaa” follows the rules of a language that has twice as many a’s as there are b’s because every time you use this rule, you add 1 “b” to the front portion of the string and the string “aa” to the back portion. Essentially, you are adding two a’s for every b in the string. If you were to pump this rule alone you would produce the string $b^i(aa)^i$ where $i \geq 0$, which will always have twice as many a’s than b’s.

<u>baSa:</u>

“baSa” follows the rules of a language that has twice as many a’s as there are b’s because every time you use this rule, you add the string “ba” to the front portion of the string and 1 a to the back portion. Essentially, you are adding two a’s for every b in the string. If you were to pump this rule alone you would produce the string $(ba)^i a^i$ where $i \geq 0$, which will always have twice as many a’s than b’s.

<u>baaS:</u>

“baaS” follows the rules of a language that has twice as many a’s as there are b’s because every time you use this rule, you add 1 b followed by 2 a’s. Essentially, you are adding two a’s for every b in the string. If you were to pump this rule alone you would produce the string $(baa)^i$ where $i \geq 0$, which will always have twice as many a’s than b’s.

---------------------------------------- Finishing Up ----------------------------------------

Since each rule follows the ruleset of the overarching language individually, concatenating the rules together in the place of the non-terminal symbol S will continue to follow the rules since no terminal symbols are removed or replaced, and exactly 2 a’s and 1 b are always added (except for of course ε which adds neither). Since every combination of these symbols are accounted for, the rules can be used to generate any and every possible string which contains twice as many a’s as b’s.  Therefore, our CFG describes the language that contains all and only strings which contain twice as many $a's$ as $b's$.