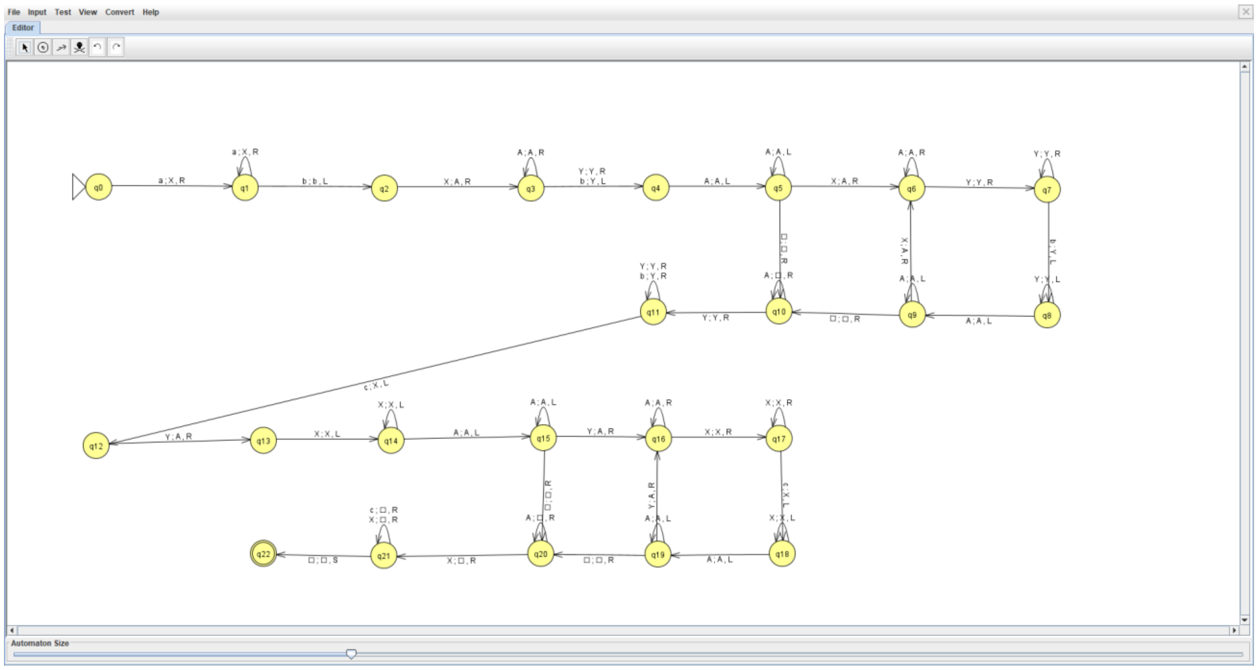
Chue Zhang

EMPL : 205042278

1a.

file included in zip for clearer picture. The approach that I took for this Turing machine was for every a there was a corresponding b ready to be read, like a pair system. If there were no more a's then we would proceed to process the rest of the b's then work on the c's and the same logic would apply. We would use the pairing system to make sure that there are more or equal amount of c's.



1b.

For abc, It will go from q0 → q1 → q2 → q3 → q4 → q5 → q10 → q11 → q12 → q13 → q14 → q15 → q20 → q21 → q22

Abc

- → Xbc
- → Abc
- → AYc
- → Yc

- ➔ YX
- ➔ AX
- ➔ X
- ➔ [] Accepted

For abbccc, It will go from q0 → q1 → q2 → q3 → q4 → q5 → q10 → q10 → q11 → q11 → q12 → q13 → q14 → q15 → q16 → q16 → q17 → q18 → q18 → q19 → q19 → q20 → q20 → q21 → q21 → q21→ q22

abbccc

- ➔ Xbbccc
- ➔ Abbccc
- ➔ AYbccc
- ➔ Ybccc
- ➔ YYccc
- ➔ YYXcc
- ➔ YAXcc
- ➔ AAXcc
- ➔ AAXXc
- ➔ AXXc
- ➔ XXc
- ➔ Xc
- ➔ C
- ➔ [] accepted

For aaaabbcccc, It will go from q0 → q1 → q1 → q1 → q1→ q2 → q3 → q4 → q5 → q6 → q6 → q7 → q8 → q8 → q8 → q9 → q6 → q6 → q6 → q7 → q7 → q7→ reject

aaaabbcccc

- ➔ Xaaabbcccc
- ➔ XXaabbcccc
- ➔ XXXabbcccc
- ➔ XXXXbbcccc

- ➔ XXXXYbcccc
- ➔ XXXAYbcccc
- ➔ XXAAYbcccc
- ➔ XXAAYYcccc
- ➔ XAAAYYcccc
- ➔ Cant find a "b" to pair up with the a so therefore there are less b's than a so we reject this string

1c.

The number of steps required is all n,m and p combined because they are the length of a,b and c (respective). Furthermore, n are paired with m's and m are paired with p's so we multiply them to get the total steps required. Its easier to understand it as in the tape, we traverse all a's then we traverse all b's then we traverse all c's

O(n + m + p + n *m + p*m)

1d.

a 3 tape turing machine would work on a b and c in their own respective tapes as opposed to our 1 tape turing machine that does everything all in one tape

2a.

In this CFG, there will never be an empty string as there is no epsilon and the only exit is by having S→ a b

a. S → a S b
b. S →a b

2b.

We broke down our CFG following the rules that A → BC, A → a

a. S → XY
b. Y → SZ
c. S → XZ
d. X → a
e. Z → b

2c.

a	a	b	b
{X}	{X}	{Z}	{Z}
N/A	{S}	N/A	
N/A	{Y}		
{S}			

For a, the only rule that could construct a is where X is present

For b, the only rule that could construct b is where Y is present

For ab, the only rule that could construct ab is S where S → X_a X_b

For abb, the only rule that could construct abb is Y where Y → SZ, then S → XZ which leaves us with XZZ which becomes abb (X → a, Z → b)

There are no rules that allow for us to create aa, bb or aab so that leaves us with just aabbb which can be constructed with S

3. Let L be a language and let there be a constant p for every string s that is in L such that there is a q and r in L where the cardinality of r is greater than or equal to c. z = xyz

1. |y| >= 1
2. |xy| <= p
3. $xy^{i}p$ where I >= 0

So we letting z = a^n b^n c^n = xyz

If we apply the third rule where I is greater than 0, then for any I that is 1 or higher would make the language not work because there would be too many a,b or c's when y is set as them.

4.

1. S → a S B C
2. S → a B C
3. S → B S C
4. S → C
5. BC → CB
6. CB → BC
7. aB → ab
8. bB → bb
9. bC → bc
10. cC → cc

Test Example below numbered with rule that was followed

abbccc

S → aSBC(1)

S → aBSCBC(3)

S → aBCCBC(4)

BC → aBCBCC(5)

CB → aBBCCC(6)

aB → abBCCC(7)

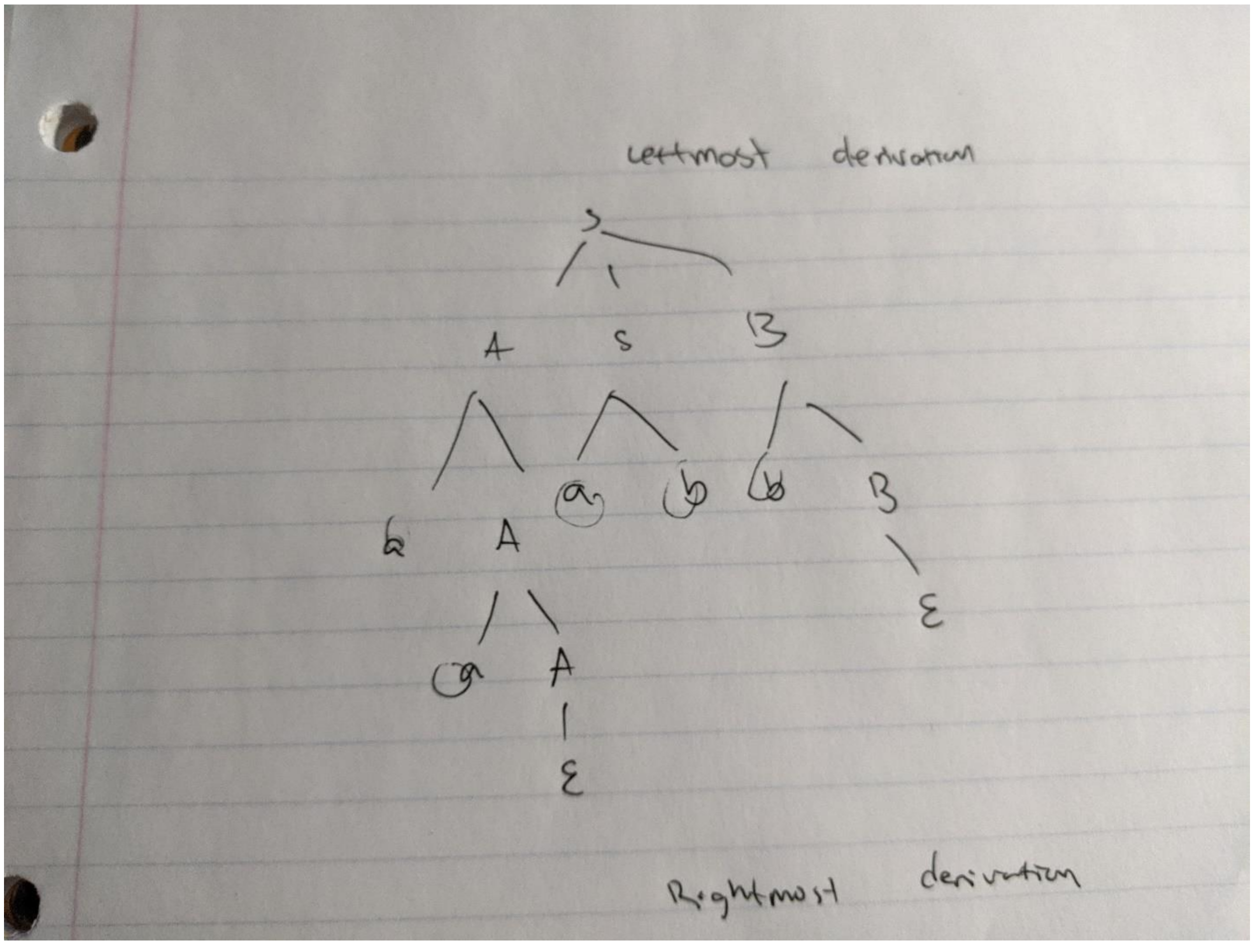
bB → abbCCC(8)

bC → abbcCC(9)

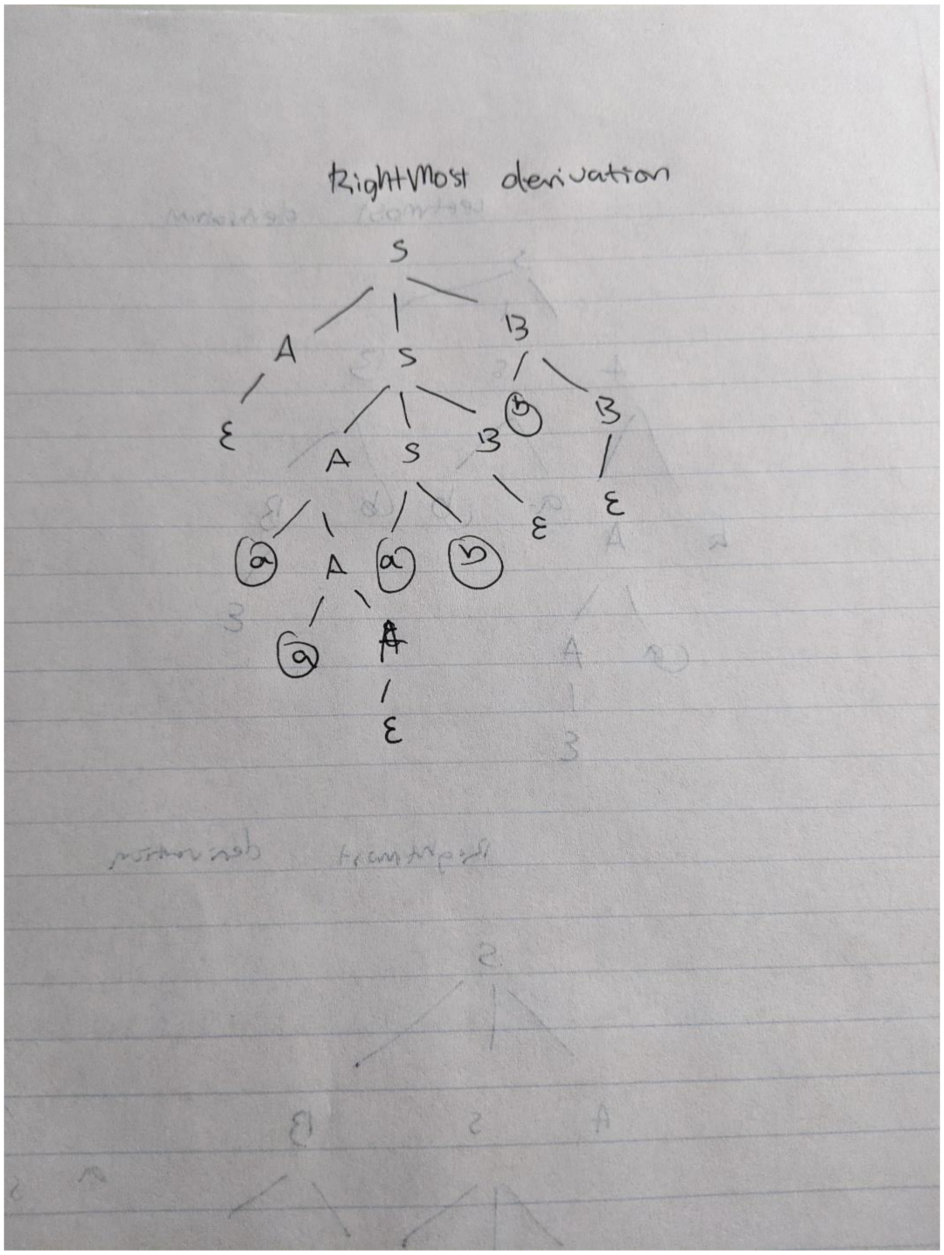
cC → abbccC(10)

cC → abbccc(10)

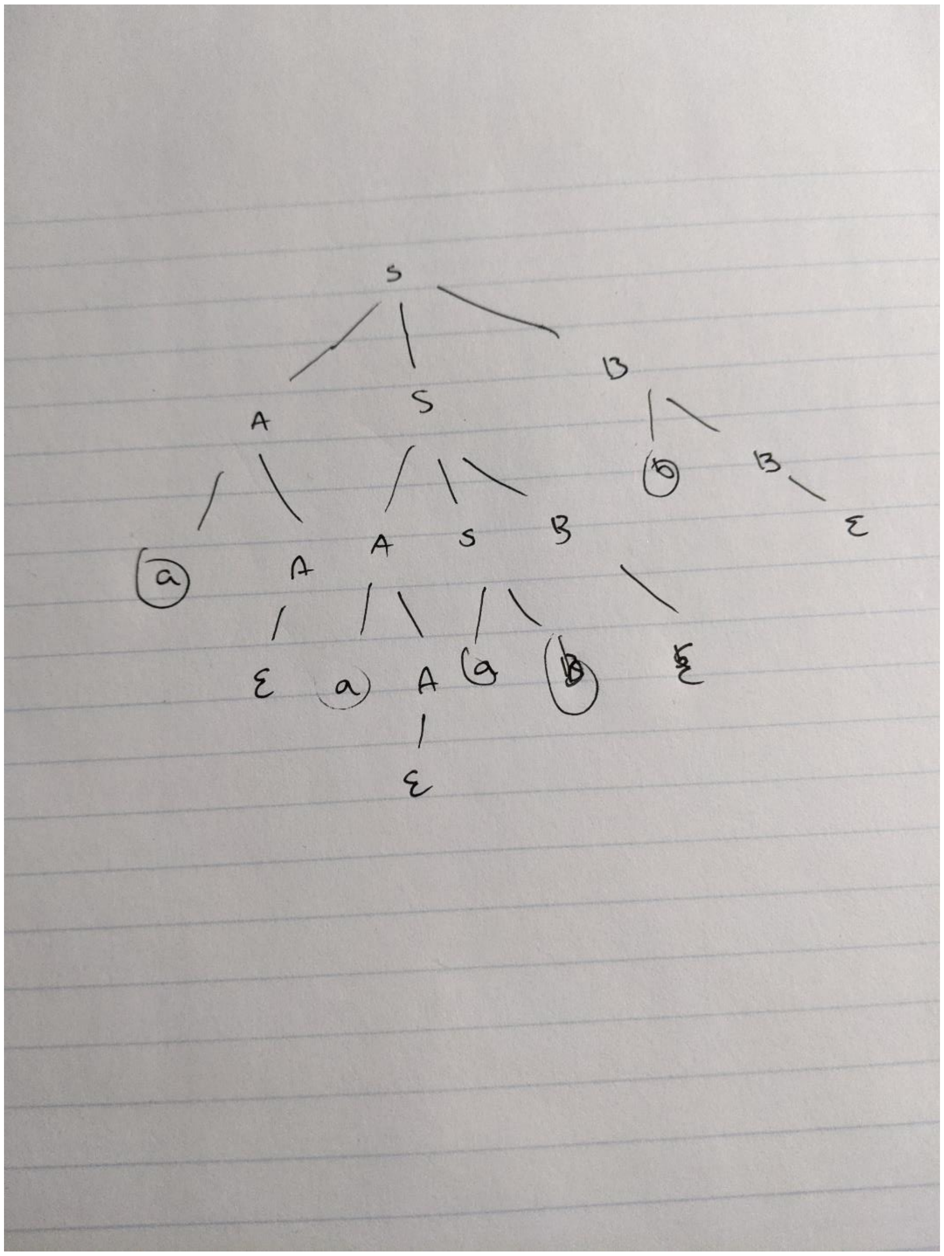
5a. This is the left most derivation where I started by looking at the a's in (aaabb) and started working with them first. The steps I took for this was I broke S → ASB where I then worked towards getting the a's first and Its required that we have at S → ab so that we don't loop infinitely. For example, if S → ASB or S → SS, it would loop infinitely unless we close out with S → ab. Next I had B → bB with the final B → epsilon.



5b. For this right most derivation, I worked with the b’s first in (aaabb) and I started out by getting as many B’s as possible before closing out with S → ab because as we stated above, its necessary to have S → ab so we don’t loop infinitely but the difference here is that I had S → ASB first so that I can get all the A's I need in the middle branch without having to consider the left branch which is why I had A → epsilon on the left.



5c. Since there is another leftmost deriviation, the grammar is ambiguous and as the definition states, for a grammar to be ambiguous, there must be 2 or more derivations of the left derivation or the right derivation and below I am showcasing another leftmost derivation.



5d.

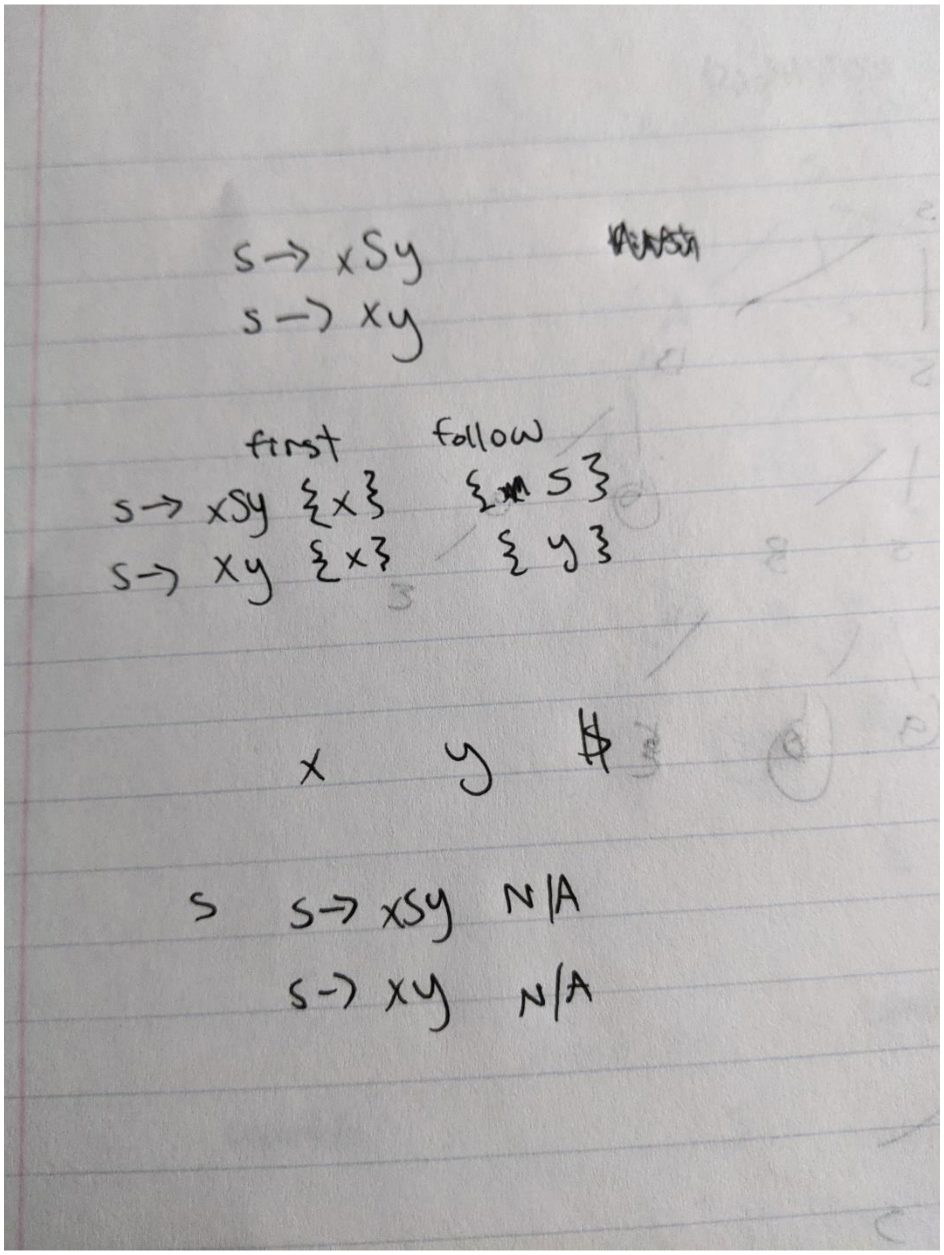
Since we remove ab, it leaves the grammar with only one way with approaching problems. Before we had to use ab because it was the only way to close the grammar and stop S from infinitely looping but now, we replaced that with an epsilon which allows for more linear approaches

$S \rightarrow ASB \mid \epsilon$

$A \rightarrow aA \mid \epsilon$

$B \rightarrow bB \mid \epsilon$

6. In the image below, please notice that the first symbol for each of S starts with x which follows up with an S or Y which is what I wrote down there. Below that is the parse table which shows what is outputted given a certain input in which there is none for Y and only has outcome for X



$S \rightarrow xSy$

$S \rightarrow xy$

	first	follow
$S \rightarrow xSy$	{x}	{S}
$S \rightarrow xy$	{x}	{y}

	x	y	$
S	$S \rightarrow xSy$	N/A	
	$S \rightarrow xy$	N/A	

7a.

Let L be a language and let there be a constant p for every string s that is in L such that there is a q and r in L where the cardinality of r is greater than or equal to c. z = xyz

4. |y| >= 1
5. |xy| <= p
6. Xy^ip where I >= 0

7b.

$W = a^nb^m$ where (n+m) >= p and m = n + 1

Based on the lemma, this is correct still because m > n

But lets choose y to generate some strings and its easy to understand that we cannot have y = “a” because otherwise, n > m. but what if we have y generate b. In the situation that I = 0 for y, that would mean that you would be removing a “b” which will lead to # of a’s = # of b’s which contradicts the lemma, therefore it is not valid as a regular language.

8. A computer scientists must be conversant with theory because it would be too much effort to determine if a piece of code works through brute force. To test all the methods out to determine if a piece of code works is too time consuming and in the end you might not even find a solution to that piece of code. That’s why its better to be conversant with theory because if you are conversant with theory, you can calculate and predict whether or not a piece of code will work or not before actually working. Think of it as planning your work out before doing the actual work. Its easier to do things when you are organized because if you go in blind and just start working, you might end up changing certain things or find out that certain things aren’t working out the way you expected them to.