Silviya Silwal
Morphology and Syntax: Assignment 2
Code description

1. **Task 1& 2**
   The CFG for all 20 sentences were created using Allen NLP constituency parsing. Below is an example of what it looks like:

```
#Muscles grow in response to work.
S -> NP VP PUNC

NP -> NNS
NNS -> 'Muscles'
VP -> VBP PP
VBP -> 'grow'
PP -> IN NP
IN -> 'in'
NP -> NP PP
NP -> NN
NN -> 'response'
PP -> IN NP
IN -> 'to'
NP -> NN
NN -> 'work'

PUNC -> '.'
```

   I loaded the CFG file using nltk.data.load() and initialized an instance of the EarleyChartParser for this grammar. I tokenized each sentence and using a *for loop* printed out the tree along with the count of number of parsing for each sentence. To count the number of parses I initialized a list and the stored the count number after every loop for each sentence. Then I divided the sum of the number of parses by the length of the sentences to get average number of parses per input sentence which was 721.66. Something to note here is that the parser for some reason was only parsing 18 of the 20 sentences. So, for that reason I excluded the two sentences when going through the loop. The two sentences were: [['Is', 'the', 'muscle', 'tired', '?'], ['Not', 'that', 'one', '.']] with index [8, 13] The grammar for these sentences start with SQ and FRAG respectively.
   For the second part of this task, I tried using the chomsky_normal_form function from nltk.tree to transform the CFG grammar to CNF but it didn't seem to work. Every time I ran the code it would process for a very long time and freeze my computer. I never got to a point where I could display any output from that code. Here is what I tried:

from nltk.tree.transforms import chomsky_normal_form
grammar_cnf = grammar_cfg.chomsky_normal_form()

   ** here grammar_cfg is the variable where CFG grammar file is loaded
   I did find a GitHub repo where someone used different class methods to convert CFG to CNF, I couldn't understand what the methods meant so I did not attempt at trying to implement it.

So, for task 2 I used the CFG file provided to us and applied all the checking methods mentioned in the assignment description.

2. INITIAL STEPS AND PROBLEMS WHEN IMPLEMENTING CKY ALGORITHM

For the CKY implementation I found this article: https://medium.com/swlh/cyk-cky-f63e347cf9b4 which gives a brief explanation about CKY algorithms. The idea I referenced from this was the *flipped_grammar* dictionary. What this method does is basically maps all the left-hand side (grammar rules) to the right-hand side (grammar productions) and vice versa. I first tried implementing the algorithm with this method, but it did not seem useful and had a lot of problems. The algorithm worked well when I filled in the diagonal elements of the table, but it seemed to run into problems when being used to fill in the rest of the rules. Hence, I discarded this idea and decided to work with productions.

During my first try, I followed the pseudo-code provided in the lecture slides to implement CKY. To do this I first created a for loop to go through every sentence in the text file. Then, I initialize an empty table using *np.zeros* , with n + 1 rows and columns each. Then I created another loop that would compare each word with the keys in the *flipped_grammar* dictionary and add it in the table if true. The I created other *for loops* as shown in the pseudo code and added the grammar rules using the *flipped_grammar* dictionary. This method however only returned correct output/tables for some sentences. This algorithm couldn't correctly find the cell it needed to populate for some of the sentences hence, this implementation turned out to be working only halfway.

I spent a lot of time trying to figure this out thinking that there might be a problem with the grammar but after testing it out with a small fake grammar that I created it seemed clear that the problem was with the way the loops were working and not the CFG grammar.

3. FINAL IMPLEMENTATION - CKY

The code for my final CKY implementation uses some logic from the pseudo code with a few changes. It also makes use of productions instead of the *flipped_grammar* dictionary implementation. In this method, I initialized the a table with n + 1 rows and columns each where each cell was a set(). Then I initialized the diagonal elements of the table with the grammar rules for each word in the sentence. Then I created three *for loops* that is used to go over to the row and column from the diagonal elements and fill them up with the grammar rules. Here, for each row and column in the table the corresponding cell k is filled by looking at the grammar rules over to its corresponding row and columns. This way we just iterate over the loops and fill in the grammar points until we reach the cell [0, n]. Once that is done and the table is filled up, I created another loop that checks whether a given sentence exists in the given grammar. If true, then it prints a message confirming the sentence is part of the grammar and if not then it gives a Checkerror message. Now, this is also a good way of knowing whether the CKY algorithm is working correctly or not because if the top right cell of the table is not filled correctly then we would get the error message and that would give us an idea about how the algorithm is not correctly filling up the table. The final loop in the code is just for printing out the sentence, it's checked verdict and the table with grammar rules. I checked the values of

the table for a few of the sentences by working on it in paper and on hindsight the output seemed to be correct. Though I would not be sure about this for all the sentences as I only checked a few of them manually.

I tried implementing the backtracking method but did not have time to finish it up as it took a lot of trial and process to implement it. I get the logic behind it but implementing it was difficult and since I spent so much time trying to fix the CKY algorithm I did not have enough time to go through the same experimentations with the backtracking. However, the basic idea of it would be to create another table for the backtracking process that would be initialized with the Nonterminal of the rules. Then, for every update to the initial table we would update the backtracking table with the rules. This way by figuring out the terminal, left, right and end node of the grammar rules we would be able to recreate the parse Trees for each sentence.

4. CONCLUSION

The most difficult part for this assignment beside the CKY algorithm was converting the CFG grammar to CNF. Because the nltk process was not working I spent a lot of time in trying to convert the grammars manually which did not seem useful or correct. Also, while creating the table for the CKY it was difficult to confirm whether the rules were being updated correctly which brought a lot of doubts to mind about whether I was implementing the method the way it was meant to not. The backtracking seemed easy by logic and I though it used some of the same implementation as the Trie structure assignment but I wasn't able to complete that part.

RESOURCES:

1. http://demo.clab.cs.cmu.edu/11711fa19/recitations/recitation2-slides-19fa.pdf
2. https://www.borealisai.com/en/blog/tutorial-15-parsing-i-context-free-grammars-and-cyk-algorithm/
3. https://courses.engr.illinois.edu/cs447/fa2018/Slides/Lecture09.pdf
4. https://markgw.github.io/uh-nlp19/day4/
5. https://medium.com/swlh/cyk-cky-f63e347cf9b4
6. https://demo.allennlp.org/constituency-parsing/