MEMORANDUM

To:	Dr. Foaad Khosmood, Lecturer, Department of Computer Science, Cal Poly SLO
	foaad@calpoly.edu
From:	Chandler Jones
	cjones87@calpoly.edu
Date:	February 17th, 2021
RE:	**Laboratory 3 for CSC 482-01**

In this memorandum, I will discuss a method for producing a Probabilistic Context Free Grammar (PCFG) from the Penn Treebank. From the source,

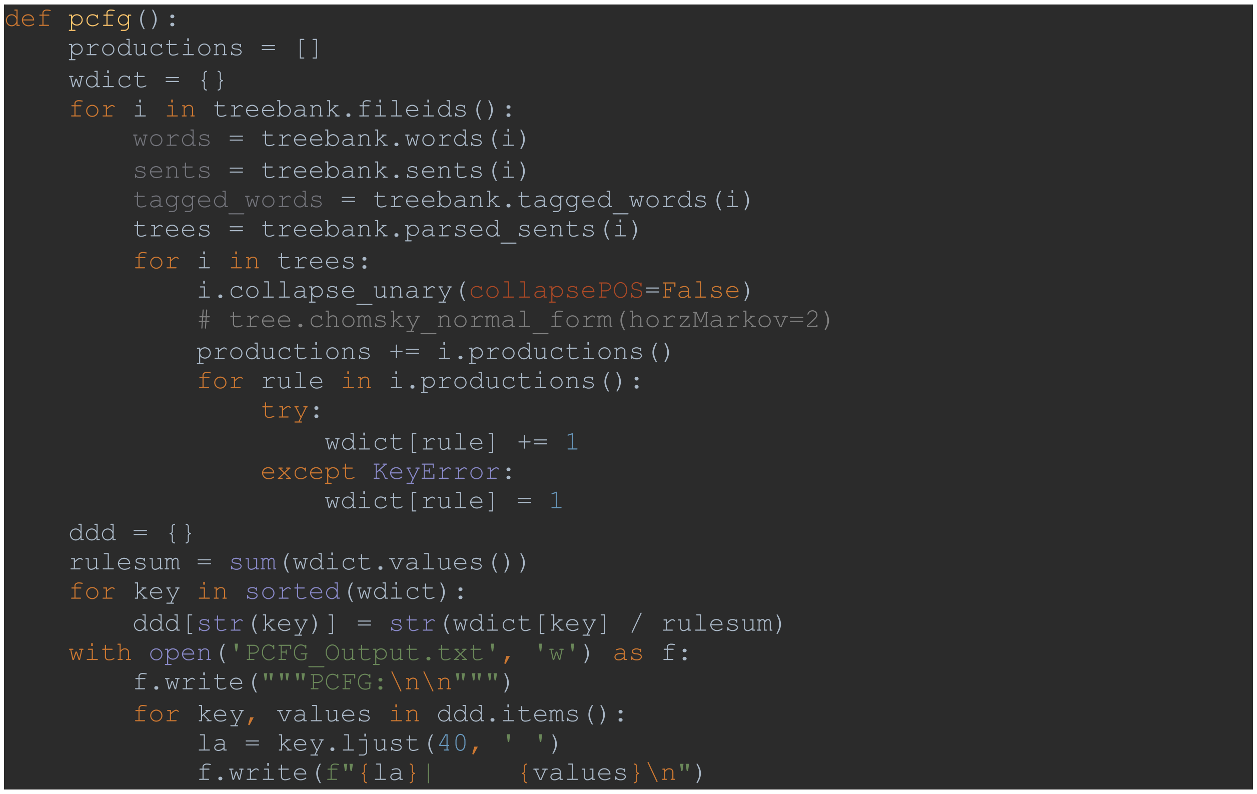
> "The Penn Treebank, in its eight years of operation (1989-1996), produced approximately 7 million words of part-of-speech tagged text, 3 million words of skeletally parsed text, over 2 million words of text parsed for predicate argument structure, and 1.6 million words of transcribed spoken text annotated for speech disfluencies. The material annotated includes such wide ranging genres as IBM computer manuals, nursing notes, Wall Street Journal articles, and transcribed telephone conversations, among others."

The probabilistic grammar method outlined below aims to develop a probability that a specific form of sentence occurs, as well as the subcomponents of that system, such as noun phrases, adjectives, and verbs, among others. This is possible in large part due to the painstaking manual process performed by the Researchers at the University of Pennsylvania, Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor.

In this brief paper, I will go on to describe the workings of the three methods performed, a word-for-word PCFG, from here on out referred to as "my PCFG", a modestly condensed PCFG, deemed "Vanilla", and an extremely condensed PCFG, deemed "Even_Further". Similarly, I will discuss the lexicalization of my PCFG, which consists of adding the corresponding head word, i.e. the key word, to each head rule within the grammar. This will serve to aid the reader in understanding the frequency of specific sentences, phrases, and words as they appear within the Penn Treebank.

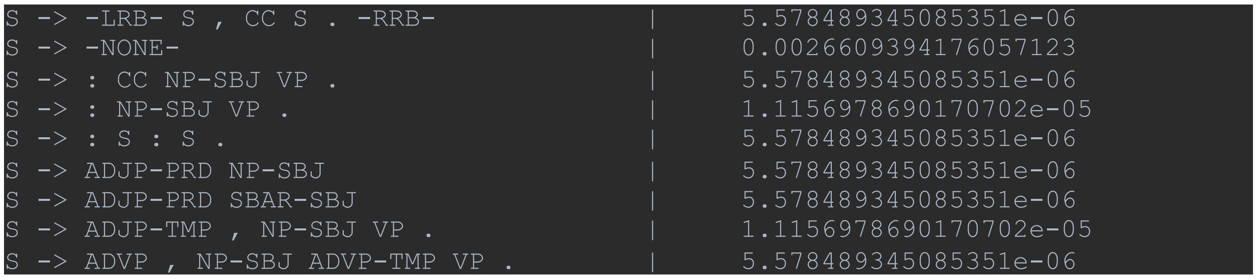
In concordance with the assignment, I will attempt to distinguish the usefulness of this method over a general CFG, and describe the efficiency of my own method.

Below in **Figure 1.** we see the initial code for constructing a PCFG. It works well in that it simultaneously generates the productions and then attempts to add them to a dictionary. If a given rule is already in the PCFG, it instead adds to the count for that given rule. Later, the values are summed, and each rule value is divided by the total sum of the rules. This is all fast. For further analysis of the different grammars, I sorted the results alphabetically, and converted them to strings before outputting to a text file. I added formatting for ease of readability, shown below in **Figure 2.**



```
def pcfg():
    productions = []
    wdict = {}
    for i in treebank.fileids():
        words = treebank.words(i)
        sents = treebank.sents(i)
        tagged_words = treebank.tagged_words(i)
        trees = treebank.parsed_sents(i)
        for i in trees:
            i.collapse_unary(collapsePOS=False)
            # tree.chomsky_normal_form(horzMarkov=2)
            productions += i.productions()
            for rule in i.productions():
                try:
                    wdict[rule] += 1
                except KeyError:
                    wdict[rule] = 1
    ddd = {}
    rulesum = sum(wdict.values())
    for key in sorted(wdict):
        ddd[str(key)] = str(wdict[key] / rulesum)
    with open('PCFG_Output.txt', 'w') as f:
        f.write("""PCFG:\n\n""")
        for key, values in ddd.items():
            la = key.ljust(40, ' ')
            f.write(f"{la}|      {values}\n")
```

Figure 1. Code for developing PCFG



```
S -> -LRB- S , CC S . -RRB-             |      5.578489345085351e-06
S -> -NONE-                             |      0.0026609394176057123
S -> : CC NP-SBJ VP .                   |      5.578489345085351e-06
S -> : NP-SBJ VP .                      |      1.1156978690170702e-05
S -> : S : S .                          |      5.578489345085351e-06
S -> ADJP-PRD NP-SBJ                    |      5.578489345085351e-06
S -> ADJP-PRD SBAR-SBJ                  |      5.578489345085351e-06
S -> ADJP-TMP , NP-SBJ VP .             |      1.1156978690170702e-05
S -> ADVP , NP-SBJ ADVP-TMP VP .        |      5.578489345085351e-06
```

Figure 2. Sample Output of standard PCFG

As you can see above in **Figure 2.**, I have organized the output neatly and sorted alphabetically. I have the probabilities distinctly separated from their corresponding rules on the right. In this output for the standard PCFG, we can see that -NONE- & -LRB- are retained, I think this may be important as they were easy to lose to simple dashes in the condensed versions.

In the Vanilla outputs below in **Figure 3.** a similar process brings us to the text results. We can see that the dashes above have been eliminated, and only the relevant characteristics remain. Additionally, we can see on the following page on **Table 1.** that indeed the number of distinct rules in this ruleset have shrunk.

```
S -> LRB S , CC S . RRB                 |      5.578489345085351e-06
S -> LST INTJ , NP VP PRN               |      5.578489345085351e-06
S -> LST NP VP .                        |      2.7892446725426755e-05
S -> LST NP VP PRN                      |      5.578489345085351e-06
S -> LST PP , NP VP .                   |      1.1156978690170702e-05
S -> LST S CC S PRN                     |      5.578489345085351e-06
S -> NONE                               |      0.002672096396295883
```

Figure 3. Sample of First Condensed "Vanilla" Output

Figure 4. below shows how some Regular Expressions have allowed us to quickly reformat and condense our rules. By completing these regex analyses before forming the dictionary, we can quickly and accurately sort the rules by common occurrence, and similarly sum/divide their values to get a probability.

```
def vanilla():
    wdict = {}
    for i in treebank.fileids():
        trees = treebank.parsed_sents(i)
        for i in trees:
            i.collapse_unary(collapsePOS=False)
            # tree.chomsky_normal_form(horzMarkov=2)
            productions = i.productions()
            for rule in productions:
                rule = str(rule)
                rule = re.sub('-NONE-', 'NONE', rule)
                rule = re.sub('\s-([A-Z]+)-', ' \\1', rule)
                rule = re.sub('[-_*+^][A-Z]{2,}|[-_*+=^][0-9]+|[-_*+^][A-Z]\b',
                              '', rule)
                try:
                    wdict[rule] += 1
                except KeyError:
                    wdict[rule] = 1

    ddd = {}
    rulesum = sum(wdict.values())
    for key in sorted(wdict):
        ddd[str(key)] = str(wdict[key] / rulesum)
    with open('Vanilla_Output.txt', 'w') as f:
        f.write("""PCFG:\n\n""")
        for key, values in ddd.items():
            la = key.ljust(40, ' ')
            f.write(f"{la}|      {values}\n")
        f.write("""\n\nCODE:\n\n""")

    return
```

Figure 4. Code to Derive "Vanilla" Output

Table 1. Number of Distinct Rules in Each Method

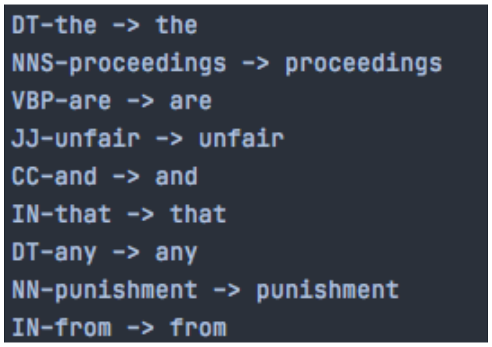
My PCFG	21821
My Vanilla	17146
My Even_Further	16754

Table 2. Time To Compute PCFG in Each Method

My PCFG	10.23 seconds
My Vanilla	9.35 seconds
My Even_Further	11.94 seconds

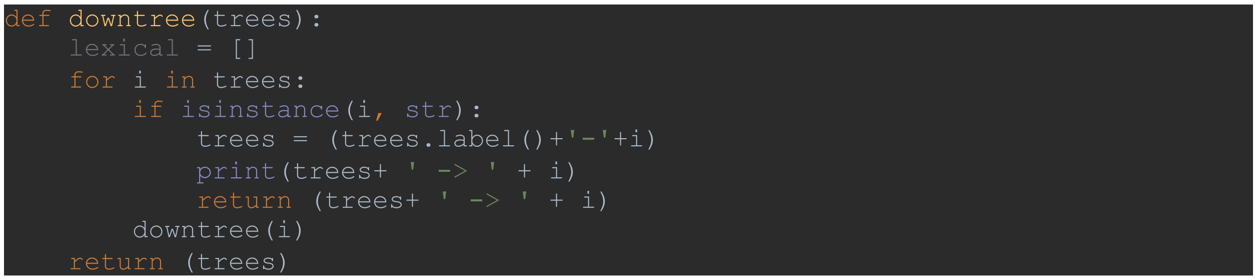
From **Table 2.** We can see that the relative time to form each of these grammars is relatively similar. We can suppose that the favorability of using these methods to conduct Grammar Rule formation is vastly superior to using conventional grammars as these are able to form many multitudes more rules, thus eliminating more unknown sentence structures, as well as providing an avenue for diverse text formation if we were to begin producing computer generated language.

For the final lexicalization of the PCFG, I am able to provide the sample shown below in **Figure 5.** Which shows the beginnings of the lexicalization. I was unable to propagate the new forms to the top, as I was merely adding strings to the .labels(), as shown in **Figure 6.** One great thing I learnt during this troubleshooting process was the dir() command. It quickly shows the possible commands you can use to alter a given element. My goal is to get more practice with recursion, that slowed me down today.



```
DT-the -> the
NNS-proceedings -> proceedings
VBP-are -> are
JJ-unfair -> unfair
CC-and -> and
IN-that -> that
DT-any -> any
NN-punishment -> punishment
IN-from -> from
```

Figure 5. Sample Recursive Output



```
def downtree(trees):
    lexical = []
    for i in trees:
        if isinstance(i, str):
            trees = (trees.label()+'-'+i)
            print(trees+ ' -> ' + i)
            return (trees+ ' -> ' + i)
        downtree(i)
    return (trees)
```

Figure 6. Existing Recursive Code