LINGUIST 492B: HW5

In this assignment, you will work with CFGs to model natural language syntax.

The primary goals of this homework are to give you practice on:

- Working with CFGs
- Thinking about how to construct grammars for natural language to capture syntactic generalizations.

Credit where it's due: This lab comes in large part from Professor Gaja Jarosz' HW5 assignment in LINGUIST492B, which is in turn adapted from an assignment by Jason Eisner (JHU).

Deliverables: Please turn 1) Commented grammar1.txt (Part 1), 2) commented grammar2.txt (Part 2), the final version of `randsent.py` (Part 3), and the answers to all questions from parts 1-4.

Part 0: Understanding the stub

As part of this lab, you have a stub `randsent_stub.py`. This implements a random sentence generator. When you run it, it will read the CFG in the `grammar.txt` file. To use this file, you can run the stub from the command line, with the first argument the name of the grammar file, and the second argument, the number of sentences to generate.

```
- python randsent_stub.py grammar.txt 5
```

You will see sentences sort of like these (maybe longer!):

```
- every pickle understood every chief of staff .
- the president understood a sandwich under every pickle !
- every sandwich with every sandwich ate a sandwich .
```

Your program works with any grammar file in the same format as `grammar.txt`, and assumes that all sentences are generated from ROOT. Notice the grammar file allows comments: any line

beginning with `#` is ignored by the program. In generating a random sentence, every non-terminal is randomly rewritten, that is, a rule that rewrites the non-terminal is randomly selected from among the rules in the grammar that have that non-terminal as a left-hand side. In the stub, each possible expansion (right-hand side) has the same probability. For example, there are two expansions to NP, meaning each has a probability of 0.5.

Make sure you understand the stub before moving on

The grammar is implemented as a dictionary with left-hand side nonterminals as keys, and a list of right-hand sides as values. Selection a random expansions on some nonterminal means choosing an element randomly from the list of right-hand-sides in the dictionary indexed by that key. The random sentence generation code is written as a recursive function with two arguments, the nonterminal to be expanded, and the grammar. It returns a string (the sentence). For each symbol that a nonterminal expands into, the function appends the symbol to the output string if it is a terminal and otherwise it appends the result of calling itself on the symbol (recursion).

Part 1: Changing the expansion probabilities

In the first part, you will extend the program to generate rules with unequal probabilities. Before doing so, though, first answer these two questions:

1. Why does the stub generate so many long sentences? Say which grammar rule is responsible, and why. What is special about this rule?
2. The grammar allows multiple adjectives, as in ”the fine perplexed pickle.” Why do your program’s sentences do this so rarely?

Next, modify the generator so that it can pick rules with unequal probabilities. Specifically, the number before the rule should represent the relative odds of picking that rule. For example, in the grammar:

```
- 3 NP A B
- 1 NP C D E
- 1 NP F
```

The three NP rules have relative odds of 3:1:1, so your generator should pick them respectively 60%, 20%, and 20% of the time. **Be careful: These numbers are more like counts and they are not probabilities since they do not sum to one.**

1. Which numbers must you modify to fix the problems in 1/2 above, to make the sentences shorter and the adjectives more frequent? Check your answer by generating some new

example sentences.
2. What other numeric adjustments can you make to the grammar in order to favor more natural sets of sentences? Experiment. Discuss at least two adjustments you made - why are they necessary and what is their effect?

HINT: the easiest way to implement this is to append as many copies of each RHS as the weight of that rule in the grammar. So, in the above example [[‘A’,’B’]] would be appended three times to the LHS ‘NP’. If you do it this way, you only have to write/modify two lines of code in the readGrammarFile() function.

Hand in your revised grammar file in a file named ‘grammar1.txt’, with comments that motivate your changes, together with 10 sentences generated by the grammar.

Part 2: Expand the grammar

Modify the grammar so it can also generate the types of phenomena illustrated in the following sentences. You want to end up with a single grammar that can generate all of the following sentences as well as grammatically similar sentences.

- Sally ate a sandwich .
- Sally and the president wanted and ate a sandwich .
- the president sighed .
- the president thought that a sandwich sighed .
- that a sandwich ate Sally perplexed the president .
- the very very very perplexed president ate a sandwich .
- the president worked on every proposal on the desk .

While your new grammar may generate some very silly sentences, it should not generate any that are obviously ungrammatical. For example, your grammar must be able to generate sentence (d) but not:

- the president thought that a sandwich sighed a pickle .

Since that is not okay English. Write rules that are as general and simple as possible. Think of each sentence above as an example of a type of construction that your grammar should account for, not as a particular sequences of words you should generate. For example, (b) is an invitation to think about how conjunctions (and, or) should be incorporated into the grammar as a whole, not just for those particular words and phrases. One of the most important things to consider when writing rules is for each verb to identify its possible frames, what kind and how many arguments it takes – such selectional restrictions must be correctly encoded in the grammar for

it to generate only grammatical sentences.

Briefly discuss your modifications to the grammar. Hand in the new grammar (commented) as a file named ‘grammar2.txt’ and 10 random sentences that illustrate your modifications.

Part 3: Exploring ambiguity

The sentence generator from the original grammar can produce the following sentence:

```
- every sandwich with a pickle on the floor wanted a president .
```

This sentence shows that the original grammar is ambiguous, because it could have been derived in either of two ways.

1. What are the two derivations (parses) for this sentence, under the grammar? You may draw trees or give a bracketed string, whichever you prefer.
2. Is there any reason to care which derivation was used? (HINT: Consider the sentence's meaning)