LINGUISTICS 492
HOMEWORK 5

INTRODUCTION

I've adopted this homework from an assignment Jason Eisner uses/used in his NLP course at Johns Hopkins. This homework will help you understand how CFGs work and how they can be used—sometimes naturally, sometimes not—to describe natural language. It will also make you think about some syntactic phenomena that are interesting in their own right.

THE STUB

The provided stub implements a random sentence generator. Each time you run the generator, it reads the (context-free) grammar from the file 'grammar.txt' and prints one or more random sentences. It takes as its first argument the name of the grammar file. The second argument is a number indicating how many random sentences to generate:

```
python randsent_stub.py grammar 5
```

Running on 'grammar.txt', you should get sentences like these:

```
the president ate a pickle with the chief of staff .

is it true that every pickle on the sandwich under the floor
understood a president ?
```

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 that non-terminal is randomly selected from among the rules in the grammar that expand that non-terminal. The stub assumes that each applicable rule has an equal chance of applying. For example, there are two rules for rewriting NP in 'grammar.txt', and each is chosen 50% of the time.

Make sure you understand how the stub works 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 the value. Selecting a random expansion of 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

In this part of the assignment, you will extend you program to generate rules with unequal probability. First answer questions 1-2 below about the stub, then implement the extension described below, and finally answer questions 3-4.

1. Why does the stub generate so many long sentences? Specifically, what 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?

Modify your generator so that it can pick rules with unequal probabilities. The number before a rule now denotes 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%, 20% of the time. Be careful, these numbers are more like counts – they are not probabilities (they don't sum to one for each left-hand-side).

3. Which numbers must you modify to fix the problems in (1) and (2) above, making the sentences shorter and the adjectives more frequent? (Check your answer by running your new generator!)
4. 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

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.

a) Sally ate a sandwich .
b) Sally and the president wanted and ate a sandwich .
c) the president sighed .
d) the president thought that a sandwich sighed .
e) that a sandwich ate Sally perplexed the president .
f) the very very very perplexed president ate a sandwich .
g) 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

In this part you will extend the program to print trees instead of sentences. Once you do this, the option "-t" will make it draw trees instead of strings when it is on:

```
python randsent.py -t mygrammar 5
```

instead of just printing:

```
The floor kissed the delicious chief of staff .
```

it should print the more elaborate version:

```
(ROOT (S    (NP   (Det  the) (Noun floor)) (VP      (Verb kissed) (NP (Det
the) (Noun (Adj  delicious) (Noun chief of staff))))).)
```

which includes extra information showing how the sentence was generated. This is a way of representing trees in text format. Each constituent is surrounded by parentheses (except the terminals, which are constituents of length 1), and the label of the constituent is the first symbol inside the constituent. Don't worry about getting the whitespace right (see below); just be sure to print the non-terminals in the right order with the right number of parentheses in the right places.

In addition to printing this bracketed string, the stub uses an NLTK module to draw a graphical representation of the tree, which you can use to visualize and check your bracketing.

HINT: you will need to reference the boolean variable 'T' in the *genNonTerminal()* function. When T is True, you will need to append additional information onto 'result'. When T is False, this function should behave just as before.

Generate about 5 more random sentences, in bracketed format. Submit them as well as the commented code for your final program as **randsent.py**.

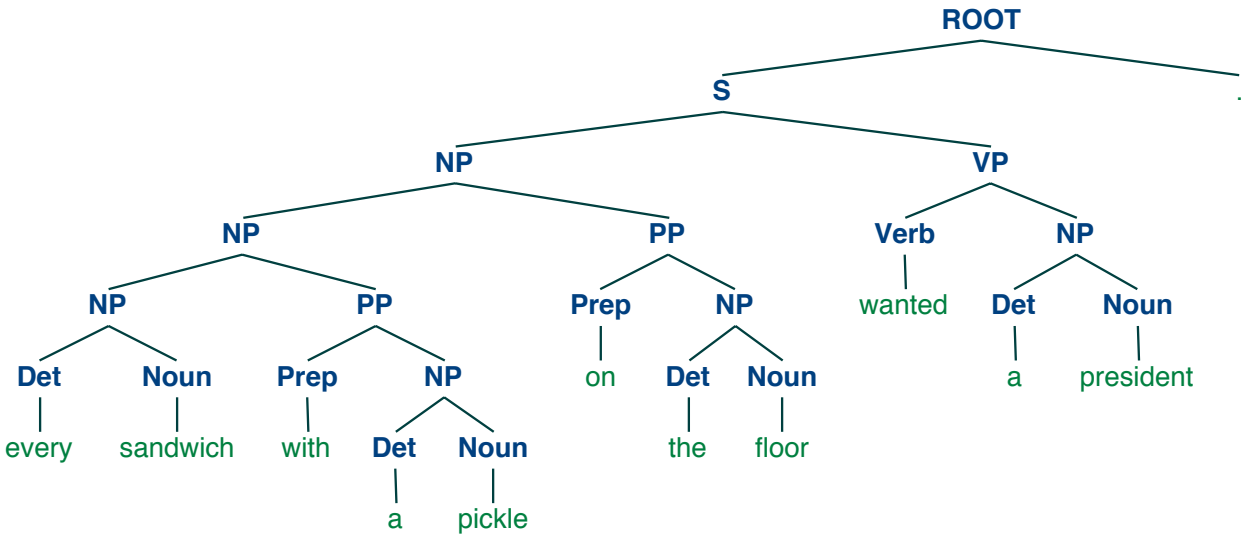
PART 4

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.

5. One derivation is as follows; what is the other (give your answer as a bracketed string or draw a tree)?



6. Is there any reason to care which derivation was used? (Hint: Consider the sentence's meaning.)

EXTRA CREDIT

Consider some other phenomenon we've talked about in class or covered in the readings, such as agreement, case, relative clauses, questions. Choose one, give some sample sentences of the phenomenon, and modify your grammar to account for this phenomenon. Check your work by generating random sentences from your new grammar.

Hand in your grammar (commented) as a file named 'grammarEC.txt'.

WHAT TO HAND IN

- Commented grammar1.txt (from Part 1)
- Commented grammar2.txt (from Part 2)
- the final version of randsent.py from Part 3
- answers to questions from Parts 1-4.