Massachvsetts Institvte of Technology
Department of Electrical Engineering and Computer Science
6.863J/9.611J, Natural Language Processing
Assignment 2: Designing Context-Free Grammars
Handed out: September 10, 2012 **Due: September 28, 2012**

Goals of the Assignment

This assignment will help you understand how context-free grammars (CFGs) work and how they can be used–sometimes comfortably, sometimes not–to describe natural language. It will also introduce you to some of the tools you will be using in the "Competitive Grammar Writing" exercise, and will also make you think about some linguistic phenomena that are interesting in their own right. In fact, there is a striking kind of 'schizophrenia' with regard to the position of CFGs in theories of how people represent, learn, and use language (which is to say, linguistic theories) as opposed to computer systems for language. In linguistic theory, as we'll discuss in class, the notions of context-free grammars have in many cases largely disappeared, leaving only the barest flicker remaining, like the grin on a Cheshire cat. In contrast, CFGs, especially their probabilistic versions, have proved exceptionally valuable in building practical parsers for natural language processing. For this assignment, the reading references on context-free grammars from the the course webpage will be helpful as background. See, e.g., JM chapter 13, and MS ch. 12.
On programming and on getting programming help: Note that you will be required to write some (small) amount of code for this assignment, as well as to think through some algorithmic issues. Since this is an NLP class, not a programming class, I don't want you wasting time on these low-level issues like how to handle I/O or hash tables of arrays If, however, you find yourself bogged down with coding or output formatting issues, then by all means seek help from someone who knows the language better! Your responsibility is the NLP stuff—you do have to design, write, and debug the interesting code **on your own**. But I don't consider it cheating if another hacker (or the TA) helps you with your I/O routines or warning messages.
How to hand in your work: Please email your answers and discussion in either ASCII text form or a pdf file as usual to `6.863-graders@mit.edu`. We will need to be able to run your code, so please either include your code as a separate zip file or point us to a directory (for example, your Athena public directory) where the zip file exists. Please insert comments in your code files so that what you have done can receive partial credit. Note that we have highlighted all the specific questions you must answer in **bold** font below. We have also supplied a LaTeX *answer template* with all the questions extracted out into a shorter form, and accompanying .sty files for it, available on the web as follows (or you can just click on the links on the course web page):

`http://web.mit.edu/6.863/fall2012/writeups/assignment2-writeup.zip`.

Initial Preparation: The files for this assignment that you may need locally can be downloaded from:

`http://web.mit.edu/6.863/fall2012/code/assignment2.zip`

This will unpack into just three files: a small `grammar` file, a script to assist in printing parse trees (which you may not even need to use), and a tiny corpus of just two sentences, to be used later on in this assignment. You can work entirely on Athena if you wish. In that case, it is preferable to copy the `zip` file to your own directory and unpack it there. Note that the latter part of this assignment, from Question 6 on, will require you to work on Athena this way in any case.

1. Write a random sentence generator. Each time you run the generator, it should read a (context-free) grammar from a file and print one or more random sentences. It should take as its first argument a path to the grammar file. If its second argument is present and is numeric, then it should generate that many random sentences, otherwise defaulting to one sentence. Name the program `randsent` so it can be run, e.g., as follows:

```
./randsent grammar 5
```

That's exactly what the graders will type to run your program, so make sure it works on Athena machines.[1]

Use the small grammar file `grammar` you downloaded earlier, and use it to test your generator. 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 ?
```

The format of the grammar file is as follows:

```
# A fragment of the grammar to illustrate the format.
1       START   ROOT
1       ROOT    S .
1       S       NP VP
1       NP      Det Noun      # There are multiple rules for NP.
1       NP      NP PP
1       Noun president
1       Noun chief of staff
```

corresponding to the context-free rules:
`START` → `ROOT`
`ROOT` → `S .`
`S` → `NP VP`
`NP` → `Det Noun`
`NP` → `NP PP`
`Noun` → `president`
`Noun` → `chief of staff`

Notice that a line for each grammar rule consists of three parts:

- a number (ignore this for now)
- a nonterminal symbol, called the "left-hand side" (LHS) of the rule (the name of a "phrase"; e.g. "NP" stands for "Noun Phrase")
- a sequence of zero or more terminal and nonterminal symbols, which is called the "right-hand side" (RHS) of the rule. (A terminal symbol is just a word or a string of words, that never appears on the LHS of any rule.)

For example, the RHS "`S .`" in the second rule is a nonterminal followed by a terminal (the period), while the RHS "chief of staff" in the second rule is a sequence of three terminals (not a single multiword terminal, so no special handling is needed).[2]

Again, ignore for now the number that precedes each rule in the grammar file. Your program should also ignore comments and excess whitespace. You should probably permit grammar symbols to contain any character *except whitespace and parentheses*.[3]

[1]Make it executable in the usual way, e.g., if it's a python program `randsent.py`, then rename the file to `randsent`, and make it executable by running `chmod +x randsent`.

[2]If you find the file format inconvenient to deal with, you can use another program to preprocess the grammar file into a more convenient format that can be piped into your program.

[3]Whitespace is being used here as a delimiter that *separates* grammar symbols. Many languages nowadays have a built-in "split" command to tokenize a line at whitespace. For example:
Python: `tokens = mystring.split();`
Which you probably already knew. If for some reason splitting at whitespace is hard for you, there is a way out. We will test your randsent program only on grammar files that you provide, and on other files that exactly match the format of our example, namely `number<tab>LHS<tab>RHS<newline>`, where the RHS symbols are separated by spaces.

The grammar's start symbol will be `START`, which in turn immediately expands into a nonterminal `ROOT` that denotes the root of the generated tree. Depth-first expansion is probably easiest, but thats up to you. Each time your generator needs to expand (for example) `NP`, it should randomly choose one of the `NP` rules to use. If there are no `NP` rules, it should conclude that `NP` is a terminal symbol that needs no further expansion. Thus, the terminal symbols of the grammar are assumed to be the symbols that appear in RHSes but not in LHSes.

Remember, your program should *read* the grammar from a file. It must work not only with the sample grammar, but with any grammar file that follows the correct format, no matter how many rules or symbols it contains. So your program cannot hard-code anything about the grammar, except for the start symbol, which is always `START`. (If you want to add a bit more to the generator, you could add a new argument that specifies what the start symbol ought to be, but this is not necessary here.)

Some advice. Make sure your code is clear and simple. If it isn't, revise it. The data structures you use to represent the grammar, rules, and sentence under construction are up to you. But they should probably have some characteristics:

- dynamic, since you don't know how many rules or symbols the grammar contains before you start, or how many words the sentence will end up having.
- good at looking up data using a string as index, since you will repeatedly need to access the set of rules with the LHS symbol youre expanding. Hash tables might be a good idea.
- fast (enough). Efficiency isn't critical for the small grammars we'll be dealing with, but its instructive to use a method that would scale up to truly useful grammars, and this will probably involve some form of hash table with a key generated from the string.

Don't hand in your code yet since you will improve it in problems 2c and 4 below. **But hand in the output of a typical sample run of 10 sentences.**

2. (a) **Why does your program generate so many long sentences?** Specifically, what grammar rule is responsible and why? What is special about this rule?

   (b) The grammar allows multiple adjectives, as in, `the fine perplexed pickle`. **Why do your program's generated sentences exhibit this so rarely?**

   (c) 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
   3.141 X NP NP
   ```

   the three NP rules have relative odds of 3:1:1, so your generator should pick them respectively $\frac{3}{5}$, $\frac{1}{5}$, and $\frac{1}{5}$ of the time (rather than $\frac{1}{3}$, $\frac{1}{3}$, and $\frac{1}{3}$ as before). Be careful: while the number before a rule must be positive, notice that it is not in general a probability, or an integer. Don't hand in your code yet since you will improve it in problem 4 below.

   (d) **Which numbers must you modify to fix the problems in 2(a) and 2(b), making the sentences shorter and the adjectives more frequent? (Check your answer by running your new generator and show that they work.)**

   (e) What other numeric adjustments can you make to the grammar in order to favor more natural sets of sentences? Experiment. **Hand in your grammar file as a file named `grammar2`, with comments that motivate your changes, together with 10 sentences generated by the grammar.**

3. **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 3d but not, e.g.:[4]

```
the president thought that a sandwich sighed a pickle .
```

Again, while the sentences should be okay structurally, they don't need to really make sense. You dont need to distinguish between classes of nouns that can eat, want, or think and those that can't.

An important part of the problem is to generalize from the sentences above. For example, question 3b is an invitation to think through the ways that conjunctions ("and," "or") can be used in English. Question 3g is an invitation to think about prepositional phrases ("on the desk," "over the rainbow", "of the United States") and how they can be used. In particular, it may not be as easy as you think to accommodate sentences such as 3(b). Think carefully.

Briefly discuss your modifications to the grammar. Hand in the new grammar (commented) as a file named `grammar3` and about 10 random sentences that illustrate your modifications.

Note: The grammar file allows comments and whitespace because the grammar is really a kind of specialized programming language for describing sentences. Throughout this assignment, you should strive for the same level of elegance, generality, and documentation when writing grammars as when writing programs.

Hint: When choosing names for your grammar symbols, you might find it convenient to use names that contain punctuation marks, such as `V_intrans` or `V[-trans]` for an intransitive verb.

4. Give your program an option "`-t`" that makes it produce trees instead of strings. When this option is turned on, as in:

```
        ./randsent -t mygrammar 5
```

instead of just printing:
`The floor kissed the delicious chief of staff .`
it should print the more elaborate version:

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

[4]Technically, the reason that this sentence is not okay is that "sighed" is an intransitive verb, meaning a verb that's not followed by a direct object. But you don't have to know that to do the assignment. Your criterion for should simply be whether it sounds okay to you (or, if you're not a native English speaker, whether it sounds okay to a friend who is one). Trust your own intuitions here, not your writing teacher's dictates.

which includes extra information showing how the sentence was generated. For example, the above derivation used the rules Noun → floor and Noun→ Adj Noun, among others.

Generate about 5 more random sentences, in tree format. Submit them as well as the commented code for your program.

Hint: You dont have to represent a tree in memory, so long as the string you print has the parentheses and nonterminals in the right places.

Hint: Its not too hard to print the pretty indented format above. But its not necessary. If your `randsent -t` just prints a simpler output format like:

```
(START (ROOT (S (NP (Det the) (Noun floor)) (VP (Verb kissed) ...
```

then you can adjust the whitespace simply by piping the output through the script `prettyprint` that you downloaded earlier:

```
./randsent -t mygrammar 5 | ./prettyprint
```

5. When I ran my sentence generator on `grammar`, it produced the sentence:

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

   This sentence is ambiguous according to the grammar, because it could have been derived in either of two ways.

   (a) One derivation is as follows; **what is the other?**

   ```
   (START (ROOT (S (NP (NP (NP (Det every)
                               (Noun sandwich))
                                    (PP (Prep with)
                                        (NP (Det a)
                                            (Noun pickle))))
                               (PP (Prep on)
                                   (NP (Det the)
                                       (Noun floor))))
                            (VP (Verb wanted)
                                (NP (Det a)
                                    (Noun president))))
                   .))
   ```

   (b) **Is there any reason to care which derivation was used?**

6. Before you extend the grammar any further, try out another tool that will help you test your grammar. It is called `parse`, and it tries to reconstruct the derivations of a given sentence – just as you did above. In other words, could `randsent` have generated the given sentence, and how? This question is not intended to be very hard – its just a chance to play around with `parse` and get a feel for whats going on, since we'll be using the same program later on in the "Competitive Grammar Writing" assignment. You will have to run this program on Athena, and get it to work with your random sentence generator.

   (a) Parsers are more complicated than generators. We will see why and how later.

   - Login to Athena, add the 6.863 locker in the usual way, and then copy the code from the assignment2 code directory to a new 6863-assignment2 directory under your home directory so that you can run `parse` from there:

```
cp -r /mit/6.863/fall2012/code/assignment2/ ~/6863-assignment2
cd ~/6863-assignment2
```

- Now try running the parser on some sentences by running the following commands in your new directory. Run the parser on some sentences by running these commands (specify the sentence to parse in single quotes, and the grammar file to use after the flag `-g`:

  ```
  echo 'the sandwich ate the perplexed chief of staff .' | ./parse -g grammar
  echo 'this sentence has no derivation under this grammar .' | ./parse -g grammar
  ```

  The first command will output a parenthesized parse tree for the first sentence, while the second command will output a sequence of warnings that the words in the input aren't possible terminals, and will simply announce `failure`. We say that the second sentence *cannot* be derived from `grammar` – that is, there is no sequence of rules, beginning with `START` and ending with the words (terminals) in the sentence. This isn't a very graceful thing to do – e.g., do you think this is what people do? Similarly, if you happen to mis-type a word, so it's not a possible terminal element, say, "egplant," then all the parser will do is complain that this is an invalid word and fail.
- The Unix pipe symbol `|` sends the output of one command to the input of another command. The following double pipe will generate 5 random sentences, send them to the parser, and then send the parses to the `prettyprint` script.

  ```
  <your-directory>/randsent grammar 5 | ./parse -g grammar|./prettyprint
  ```

  Fun, huh? Note: the parser may run very SLOWLY on Athena if a sentence is long, so be patient!
- Use the parser to check your answers to question 3. If you did a good job, then `./parse -g grammar3` should be able to parse all the sample sentences from question 3 as well as similar sentences. This kind of check will come in handy again when you tackle question 7 below. It is a VERY good way to check that your grammar is OK.
- Use `./randsent -t  5` to generate some random sentences from `grammar2` or `grammar3`. Then try parsing those same sentences with the same grammar. **Does the parser always recover the original derivation that was "intended" by `randsent`? Or does it ever "misunderstand" by finding an alternative derivation instead? Discuss. (This is the only part of question 6a that you need to answer in your writeup for this assignment.)**

(b) **How many ways are there to analyze the following Noun Phrase (NP) under the original grammar?** (That is, how many ways are there to derive this string if you start from the NP symbol of grammar?)

```
every sandwich with a pickle on the floor under the chief of staff
```

Explain your answer. Now, *check* your answer by using the argument flags to `parse`. (You can see what these are by running `./parse -h`. Note that you will need to provide 2 argument flags to answer this question. **Show that you can check your result this way as part of your write-up.**)

(c) By mixing and matching the commands above, generate a bunch of sentences from `grammar`, and find out how many different parses they have. Some sentences will have more parses than others. **Do you notice any patterns? Try the same exercise with `grammar3`.**

(d) When there are multiple derivations, this parser chooses to return only the most probable one. (Ties are broken arbitrarily.) Parsing with the `-p -b -d` options will tell you more about the probabilities. Feed the parser using `grammar` with a corpus consisting of the following two sentences, e.g., `twosent.txt`, so that you can use the `-i` argument of the parser to just input the file, without the need to type in the sentences. (Recall that this file is included in the `.zip` file for the assignment.)

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

Now run the parser on these two sentences using the probability flags:
`./parse -p -b -d -g grammar -i twosent.txt | ./prettyprint`
You should try to understand the resulting numbers, where $p(x)$ denotes the probability of a sentence, and $p(x|y)$ is the conditional probability of x given y. We will be talking about probabilities and context-free grammars in the coming weeks, but again, you can look at JM, pp. 459-464, and/or MS, pp. 381-389.

i. The first sentence reports:
   ```
   p(sentence)= 5.144032922e-05
   p(best_parse)= 5.144032922e-05
   p(best_parse|sentence)= 1
   ```
   Why is `p(best_parse)` so small? What probabilities were multiplied together to get its value of 5.144032922e-05? (*Hint*: Look at `grammar`.) `p(sentence)` is the probability that `randsent` would generate this sentence. **Why is it equal to `p(best_parse)`? Why is `p(best_parse|sentence)`=1?**

ii. The second sentence reports:
   ```
   p(sentence)= 1.240362877e-09
   p(best_parse)= 6.201814383e-10
   p(best_parse|sentence)= 0.5
   ```
   What does it mean that `p(best_parse|sentence)` is 0.5 in this case? Why would it be *exactly* 0.5? (*Hint*: Again, look at `grammar`.)

iii. Put the 2 sentences above in a file if you haven't done that already. Then run the parser with the flag `-C` and `-i <yourinputfile>`. After reading the whole 18-word corpus (including punctuation), the parser reports how well the grammar did at *predicting* the corpus, using a measure called *cross-entropy.* (See pp. 116-118 of JM or pp. 74-76 of MS). **Explain exactly how the following numbers below were calculated from the two sets of numbers above, that is, from the parse of each of the two sentences:**

   ```
   cross-entropy = 2.435185067 bits = -(-43.8333312 log-prob./18 words)
   ```

   Remark: Thus, a compression program based on this grammar would be able to compress this corpus to just 44 bits, which is $<$ 2.5 bits per word. A "deep result" is that the better a grammar, the more it can compress a corpus of sentences. (In fact, in effect a context-free grammar can compress an *infinite* corpus into a *finite* set of rules. This is what the 19th century scientist Alex Von Humboldt meant by "the infinite use of finite means" – even though our brains are finite, they evidently have *finitary* rule systems that can *generate* an infinite variety of outputs. The same is clearly true of, e.g., of the systems for addition or multiplication. However, until the advent of the 20th century and recursive function theory and the modern theory of computation, no one knew how to actually make this work.)

iv. **Based on the above numbers, what *perplexity* per word did the grammar achieve on this corpus?** (Perplexity is just a transform of cross-entropy, as your textbooks, and your lecture, explain.)

v. But the compression program might not be able to compress the following corpus that consists of just two sentences very well. **Why not? What cross-entropy does the grammar achieve this time? Try it and explain.**

   ```
   the president ate the sandwich .
   the president ate .
   ```

vi. I made up the two corpora above out of my head. But how about a large corpus that you actually generate from the grammar itself? Let's try `grammar2`: it's natural to wonder, **how well does `grammar2` do on average at predicting word sequences that it generated itself? Please provide an answer in bits per word. State the command (a Unix pipe) that you used to compute your answer.** This is called the *entropy* of `grammar2`. A grammar has high entropy if it is "creative" and tends to generate a wide variety of sentences, rather than the same sentences again and again. So it typically can generate sentences that are unlikely, even with respect to its own description of a language. **How does the entropy of your `grammar2` compare to the entropy of your `grammar3`? Discuss. Try to compute the entropy of the original grammar. What goes wrong and why?**

vii. If you generate a corpus from `grammar2`, then `grammar2` should on average predict this corpus better than `grammar` or `grammar3` would. In other words, the entropy will be *lower* than the cross-entropies. **Check whether this is true: compute the numbers and discuss.**

7. Now comes perhaps the most important question of the assignment! **Think about all of the following phenomena, and extend your grammar from question 3 to handle them. (Be sure to handle the particular examples suggested.)** As always, try to be elegant and general, but you will find that these phenomena are somewhat hard to handle elegantly with CFG notation. **Call your resulting grammar `grammar4` and be sure to include it in your write-up along with examples of it in action on new sentences like the ones illustrated below.**

   Important: Your final grammar should handle everything from question 3, plus the two new phenomena. This means you have to worry about how your rules might interact with one another. Good interactions will elegantly use the same rule to help describe two phenomena. Bad interactions will allow your program to generate ungrammatical sentences, which will hurt your grade!

   (a) *Yes-no questions.* Examples:
   ```
   did Sally eat a sandwich ?
   will Sally eat a sandwich ?
   ```
   Of course, don't limit yourself to these simple sentences. Also consider how to make yes-no questions out of the statements in question 3.

   (b) *WH-word questions.* Examples:
   ```
   what did the president think ?
   what did the president think that Sally ate ?
   what did Sally eat the sandwich with ?
   who ate the sandwich ?
   where did Sally eat the sandwich ?
   ```

We shall see in future lectures how all these rules might be learned, along with some of the complications they entail.