601.465/665 — Natural Language Processing
Assignment 1: Designing Context-Free Grammars

Prof. Jason Eisner — Fall 2018
Due date: Monday 17 September, 2 pm
PLEASE GET THIS ONE IN ON TIME!

This assignment will help you understand how CFGs work and how they can be used—sometimes comfortably, sometimes not—to describe natural language. It will also make you think about some linguistic phenomena that are interesting in their own right.

> **Collaboration:** *You may work in groups of up to 3 on this assignment.* That is, if you choose, you may collaborate with 1–2 partners from the class, handing in a single homework with multiple names on it. You are expected to do the work *together*, not divide it up: if you didn't work on a question, you don't deserve credit for it! Your solutions should emerge from collaborative real-time discussions with the whole group present.
>
> Because the assignment is about modeling English, non-native speakers of English are recommended to partner up with native speakers. (This will be instructive for both of you.) You can use Piazza to look for teammates.

Programming language: In questions 1, 2c, 2f, and 4 you will develop a single small program. I don't care what programming language you use so long as the code is commented and readable. But try to use one that will make your life easy.

(I recommend a language with good support for strings, dictionaries, and lists, so you can easily read the grammar file and store all the possible ways to rewrite a symbol like VP. Python would be a natural choice. I found that a 35-line Perl solution to the whole shebang was very quick and easy to write, whereas a Java or C solution would probably have been longer and more annoying.)

Readings: You can do this homework based on only this handout and your ingenuity. However, some students in the past have found it helpful to look at the readings listed for this week on the syllabus (http://cs.jhu.edu/~jason/465). If you find other good readings (written by actual linguists) about English grammar, please share them on Piazza. It is fine to consider what the web has to say about questions of English grammar, but this may be misleading. Of course, observe academic integrity and do not claim any work by third parties as your own.

How to hand in your work: The ✍ symbol in the left margin marks items that you need to hand in.

Besides the comments you embed in your source and grammar files, put all other notes, documentation, generated sentences, and answers to questions in a file called README.pdf. You may produce README.pdf from a TEXfile, a text file, a word processing document, or whatever you like. It may include scans of neatly handwritten formulas or hand-drawn diagrams.

README.pdf should refer to question numbers like 3(a). (Don't use the blue ✍ numbers; they are for your personal reference but may change if this assignment handout is updated.)

✍₁ You will submit README.pdf using a system called Gradescope, and you'll find it convenient to start a new page for each question you answer.

Your grammar files and executable file(s) will need to be placed into a single submission directory, ✍₂ which you will zip and upload separately on Gradescope. We'll post more detailed instructions on Piazza.

1. Write a random sentence generator. Each time you run the generator, it should read the (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 (we'll use the extension `.gr` for such files). 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 as:

   ```
   ./randsent grammar.gr 5
   ```

   That's exactly what the graders or autograder will do to run your program, so make sure it works—and works on the `ugrad` machines! If necessary, make `randsent` be a wrapper script that calls your real program. For example, if your real program is in Java, then `randsent` might be a file consisting of the single line

   ```
   java RandSent $*   # works since you submitted RandSent.class
   ```

   Or if your program is in C++, `randsent` might compile it (if needed) and then run it:

   ```
   make >/dev/null    # works since you submitted a Makefile that calls g++
   ./a.out $*
   ```

   To make this script executable so that you and the graders can run it from the command line as shown above, type

   ```
   chmod +x randsent
   ```

   Download the small grammar at http://cs.jhu.edu/~jason/465/hw-grammar/grammar.gr 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.gr` file is as follows:

   ```
   # A fragment of the grammar to illustrate the format.
   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 rules

ROOT → S .	NP → Det Noun	Noun → president
S → NP VP	NP → NP PP	Noun → chief of staff

   Notice that a line consists of three parts:

   - a number (ignore this for now)
   - a nonterminal symbol, called the "left-hand side" (LHS) of the rule
   - a sequence of zero or more terminal and nonterminal symbols, which is called the "right-hand side" (RHS) of the rule

   For example, the RHS "S ." in the first rule is a nonterminal followed by a terminal, 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!).

Again, ignore for now the number that precedes each rule in the grammar file. Your program should also ignore comments and excess whitespace.[1] You should probably permit grammar symbols to contain any character *except whitespace and parentheses and the comment symbol* `#`.[2]

The grammar's start symbol will be `ROOT`, because it is the root of the tree. Depth-first expansion is probably easiest, but that's 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, `grammar.gr`, but with *any* grammar file that follows the correct format,[3] 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 `ROOT`.

> *Advice from a previous TA:* 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 you're expanding. Hash tables might be a good idea.
> - fast (enough). Efficiency isn't critical for the small grammars we'll be dealing with, but it's 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. Python and Perl offer built-in "dictionaries," which hide all the messy details of the hash table, instead letting you use notation that looks like indexing an array with a string variable.
> - familiar to you. You can use any structure in any language you're comfortable with, if it works and produces readable code. We'll probably grade programs somewhat higher that have been designed with the above goals in mind, but correct functionality and readability are by far the most important features for grading. Meaningful comments are your friend!

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

---

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

[2] 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();`
- Perl: `@tokens = split(" ",myline);`
- Java: `String tokens[] = myline.split("\\s+");`

Hopefully you already knew that. The course assumes that you will not have to waste much time worrying about these unimportant programming issues. If you find yourself getting sidetracked by them, consider picking a different programming language. (In particular, I recommend against C/C++.) Or ask for help on Piazza.

   But 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.

[3] You can assume that the input file (and the command-line arguments) will be in the correct format. Checking this is not a requirement. Of course, it is still good programming practice to verify that an input file is in the correct format. You are preparing the grammar files by hand; an uncaught editing error in your grammar file might produce misleading or confusing behavior. So it may be wise to catch such errors—just for your own sake. (Don't obsess about this, though. You probably won't need to reuse your randsent program in future assignments; if it works well enough for this assignment, that's ok.)

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 sentences do 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 $\frac{1}{3}$, $\frac{1}{3}$, $\frac{1}{3}$ 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 questions 2f and 4 below.

(d) Which numbers must you modify to fix the problems in (a) and (b), making the sentences shorter and the adjectives more frequent? (Check your answer by running your generator!)

(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 in a file named `grammar2.gr`, with comments that motivate your changes, together with 10 sentences generated by the grammar.

(f) To be safe, modify `randsent` so that it will terminate on *any* grammar.

Simple grammars exist for which PCFG generation has probability > 0 of running forever:

```
1       S x
2       S S S
```

This grammar sometimes generates small strings like `x` or `x  x`, but its probability of terminating at all is only $\frac{1}{2}$. The rest of the time, it runs forever, generating a tree that is literally infinitely deep.[4] Furthermore, if you change the number on the second line to `1`, then the probability of terminating is 1, but the expected size of the generated tree is ∞.

Our grading system will obviously not be able to deal with trees whose size is measured in petabytes and which take years to print out. Nor will it be able to deal with infinite trees. So let's modify `randsent` so that it gracefully declines to finish a sentence that is getting too long. The most obvious solution is to limit the depth of recursion. However, a balanced binary tree of depth 50 would require more than a petabyte to print out, so even limiting the stack depth to 50 is no guarantee of reasonable-size output. (The default limit in Python seems to be 1000.)

A more effective solution: Limit the total number of nonterminals that you are willing to expand in a single sentence. Once you have called the nonterminal expansion function at least M times, future calls to it should just print "`...`". So at that point, the program can't launch any new recursive calls; it will eventually return from all of its existing calls.

How big should the limit M be? I checked a corpus of English parse trees, and the biggest one was a monster 250-word sentence with 440 nonterminals and tree depth of 11. So $M = 450$, say, would be a reasonable choice.[5]

Don't hand in your code yet since you will improve it in question 4 below.

---

[4]At least in theory—if you run your program on an idealized computer with infinite memory. Can you prove this?

[5]Notice that since your tree certainly can't be any deeper than M, these values guarantee that you'll never encounter Python's default recursion limit of 1000 (at least not with a typical implementation of `randsent`).

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) `it perplexed the president that a sandwich ate Sally .`

   (f) `that a sandwich ate Sally perplexed the president .`

   > *Note:* Yes, 3f is acceptable in standard written English. It means the same thing as 3e. Are there any other verbs that could replace `perplexed` in 3e and/or 3f?

   (g) `the very very very perplexed president ate a sandwich .`

   (h) `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 "not okay" English. For example, your grammar must be able to generate 3d but not

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

   since that sentence is "not okay." The technical term is **unacceptable**, meaning that a human native speaker of English would not *accept* the sentence as grammatical, according to the human's internal grammar ...so it shouldn't be grammatical according to your grammar either. The symbol `*` is traditionally used to mark unacceptable utterances.[6]

   Again, while the sentences should be okay structurally, they don't need to really make sense. You don't need to distinguish between classes of nouns that can eat, want, or think and those that can't.[7]

   An important part of the problem is to *generalize* from the sentences above. For example, 3b is an invitation to think through the ways that conjunctions ("and," "or") can be used in English. 3h is an invitation to think about prepositional phrases ("on the desk," "over the rainbow", "of the United States") and how they can be used.

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

   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.

---

[6]Technically, the reason that this sentence is unacceptable 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 "acceptable English" 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.

[7]After all, the following poem (whose author I don't know) is perfectly good English:

From the Sublime to the Ridiculous, to the Sublimely Ridiculous, to the Ridiculously Sublime
An antelope eating a cantaloupe is surely a strange thing to see;
But a cantaloupe eating an antelope is a thing that could never be.
And an antelope eating an antelope is a thing that could hardly befall;
But a cantaloupe eating a cantaloupe, well, that could never happen at all.

The point is that "cantaloupe" can be the subject of "eat" even though cantaloupes can't eat. It is acceptable to *say* that they can't—or even to say incorrectly that they can.

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.gr 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. For example, the above derivation used the rules Noun → floor and Noun → Adj Noun, among others.

Once your program hits the limit M, it will print "`...`" as before. For example, with $M = 4$, you'd sometimes get incomplete output like `(S (NP Sally) (VP (VP found ...) ...))`.

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

Hint: You don't have to represent a tree in memory, so long as the string you print has the parentheses and nonterminals in the right places.

Hint: In Python, the `argparse` module (`https://docs.python.org/3/howto/argparse.html`) can help you process command-line arguments, including optional arguments like `-t`.

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

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

then you can adjust the whitespace simply by piping the output through a prettyprinter:

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

Just download that `prettyprint` filter script from `http://cs.jhu.edu/~jason/465/hw-grammar/prettyprint`.

Suggestion (optional): You may also want to implement an option "`-b`" that uses occasional brackets to show only some of the tree structure, e.g.,

```
{[the floor] kissed [the delicious chief of staff]} .
```

where `S` constituents are surrounded with curly braces and `NP` constituents are surrounded with square brackets. This may make it easier for you to read and understand long random sentences that are produced by your program.

6

5. When I ran my sentence generator on `grammar.gr`, 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?

   ```
               (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? (*Hint:* Consider meaning.)

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—it's just a chance to play around with `parse` and get a feel for what's going on.

   (a) Parsers are more complicated than generators. You'll write your own parser later in the course. For now, just use one that we've installed on the `ugrad` machines:

   - Log on to one of the `ugrad` machines, and change to this directory:
     `cd /usr/local/data/cs465/hw-grammar`
   - Try running the parser by typing
     `./parse -g grammar.gr`
     You can now type sentences (one per line) to see what you get back:
     `the sandwich ate the perplexed chief of staff .`
     `this sentence has no derivation under this grammar .`
     Press Ctrl-D to end your input or Ctrl-C to savagely abort the parser.
   - 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 prettyprinter.
     `./randsent grammar.gr 5 | ./parse -g grammar.gr | ./prettyprint`
     Fun, huh?
   - Use the parser to check your answers to question 3. If you did a good job, then `./parse -g grammar3.gr` should be able to parse 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.

- Use `./randsent -t 5` to generate some random sentences from `grammar2.gr` or `grammar3.gr`.
  Then try parsing those same sentences with the same grammar.

  ☞12 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 README.)

☞13 (b) How many ways are there to analyze the following **noun phrase** under the original grammar? (That is, how many ways are there to derive this string if you start from the NP symbol of `grammar.gr`?)

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

Explain your answer. Now, *check* your answer using some other options of the `parse` command (namely `-c` and `-s`; you can type `./parse -h` to see an explanation of all the options).

☞14 (c) By mixing and matching the commands above, generate a bunch of sentences from `grammar.gr`, 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.gr`.

(d) When there are multiple derivations, this parser chooses to return only the *most probable* one. (Ties are broken arbitrarily.) Parsing with the `-P` option will tell you more about the probabilities:

```
./parse -P -g grammar.gr | ./prettyprint
```

Feed the parser a corpus consisting of 2 sentences:

```
the president ate the sandwich .
every sandwich with a pickle on the floor wanted a president .
[Ctrl-D]
```

You should try to understand the resulting numbers (after the lecture about probabilities).

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

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

  iii. After reading the whole 18-word corpus (including punctuation), the parser reports how
  ☞19 well the grammar did at predicting the corpus. Explain exactly how the following numbers were calculated from the numbers above:
```
# 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.

      iv. Based on the above numbers, what *perplexity* per word did the grammar achieve on this corpus? (Remember from lecture that perplexity is just a variation on cross-entropy.)

      v. But the compression program might not be able to compress the following corpus too well. Why not? What cross-entropy does the grammar achieve this time? Try it and explain.

```
the president ate the sandwich .
the president ate .
[Ctrl-D]
```

(e) 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*?

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 generates sentences that even it thinks are unlikely.

How does the entropy of your `grammar2` compare to the entropy of your `grammar3`? Discuss.

Try to compute the entropy of the original `grammar.gr`; what goes wrong and why?

(f) 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 the main question of the assignment! Think about all of the following phenomena, and extend your grammar from question 3 to handle ANY TWO of them—your choice. Briefly discuss your solutions and provide sample output.

Be sure you can handle the particular *examples* suggested, which means among other things your grammar must include the words in those examples.

You should also generalize appropriately beyond these examples. As always, try to be elegant in your grammar design, but you will find that these phenomena are somewhat hard to handle elegantly with CFG notation. We'll devote most of a class to discussing your solutions.

Important: Your final grammar should handle everything from question 3, **plus both** of the phenomena you chose to add. 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 unacceptable sentences, which will hurt your grade!

(a) *"a" vs. "an."* Add some vocabulary words that start with vowels, and fix your grammar so that it uses "a" or "an" as appropriate (e.g., `an apple` vs. `a president`). This is harder than you might think: how about `a very ambivalent apple`?

(b) *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.

(c) *Relative clauses.* Examples:

9

- the pickle kissed the president that ate the sandwich .
- the pickle kissed the sandwich that the president ate .
- the pickle kissed the sandwich that the president thought that Sally ate .

Of course, your grammar should also be able to handle relative-clause versions of more complicated sentences, like those in 3.

Hint: These sentences have something in common with 7d.

(d) *WH-word questions.* If you also did 7b, handle questions like

- 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 ?

If you didn't also do 7b, you are allowed to make your life easier by instead handling "I wonder" sentences with so-called "embedded questions":

- I wonder what the president thought .
- I wonder what the president thought that Sally ate .
- I wonder what Sally ate the sandwich with .
- I wonder who ate the sandwich .
- I wonder where Sally ate the sandwich .

Of course, your grammar should be able to generate wh-word questions or embedded questions that correspond to other sentences.

Hint: All these sentences have something in common with 7c.

(e) *Singular vs. plural agreement.* For this, you will need to use a present-tense verb since past tense verbs in English do not show agreement. Examples:

- the citizens choose the president .
- the president chooses the chief of staff .
- the president and the chief of staff choose the sandwich .

(You may not choose both this question and question 7a, as the solutions are somewhat similar.)

(f) *Tenses.* For example,

the president has been eating a sandwich .

Here you should try to find a reasonably elegant way of generating all the following tenses:

	present	past	future
simple	eats	ate	will eat
perfect	has eaten	had eaten	will have eaten
progressive	is eating	was eating	will be eating
perfect + progr.	has been eating	had been eating	will have been eating

(g) *Appositives.* Examples:

- the president perplexed Sally , the fine chief of staff .
- Sally , the chief of staff , 59 years old , who ate a sandwich , kissed the floor .

The tricky part of this one is to get the punctuation marks right. For the appositives themselves, you can rely on some canned rules like

$$\text{Appos} \quad \rightarrow \quad \text{59 years old}$$

although if you also did 7c, try to extend your rules from that problem to automatically generate a range of appositives such as `who ate a sandwich` and `which the president ate.`

✍26 Hand in your grammar (commented) as a file named `grammar7.gr`. Be sure to indicate clearly which TWO of the above phenomena it handles.

8. *Extra credit:* Impress us! How much more of English can you describe in your grammar? Extend `grammar7` in some interesting way (or create a wholly new grammar) and tell us about it. For ideas,
✌27 you might look at some random sentences from a magazine. Name the grammar file `grammar8.gr`.

If it helps, you are also free to extend the notation used in the grammar file as you see fit, and change
✌28 your generator accordingly. If so, name the extended generator `randsentx`.

You may enjoy looking at the output of the Postmodernism Generator, http://www.elsewhere.org/pomo, which generates random postmodernist papers. Then, when you're done laughing at the sad state of the humanities, check out SCIgen http://pdos.csail.mit.edu/scigen/, which generates random computer science papers—one of which was actually accepted to a vanity conference.

Both generators work exactly like your `randsent`, as far as I know. SCIgen says it uses a context-free grammar; the Pomo generator says it uses a recursive transition network, which amounts to the same thing. I suspect, however, that their grammars contain a lot of long canned phrases with blanks to fill in—sort of like Mad Libs (e.g., http://www.eduplace.com/tales) with academic jargon. That's probably not what you want in a general-purpose grammar of English, which is supposed to show how to *build up* those long phrases according to basic, reusable principles of English.

You might also like to try your `randsent` on some larger grammars at http://cs.jhu.edu/~jason/465/hw-grammar/extra-grammars, just for fun, or as inspiration for your own grammar.