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

Prof. Jason Eisner*— Fall 2023
Due date: Wednesday 13 September, 2pm ET

Homework goals: After completing this homework, you should be able to

- Understand how CFGs work and how they can be used to describe natural language
- Realize that natural language is complicated and describing it accurately can be tricky
- Understand how parsers use probability to disambiguate sentences
- Be comfortable programming in Python 3
  (all homeworks will be in Python 3; use this homework to catch up if you're unfamiliar)

Collaboration: ***You may work in groups of up to 3 on this homework.***

- 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 involving the whole group.
- Because the homework 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. (See post `@5`.)
- You can do this homework using only this handout and your ingenuity. However, some students in the past have found these optional readings helpful: J&M 12, M&S 3, Huddleston. If you find other good readings about English grammar (written by actual linguists), please share on Piazza.
- Remember academic integrity and do not claim any work by third parties as your own.

Starter code: The following starter code is available in `http://cs.jhu.edu/~jason/465/hw-grammar`:

- `randsent.py`
- `prettyprint`
- `grammar.gr`
- `parse`

How to hand in your work: This homework has coding and written components. The code components will be filled out in `randsent.py` and any created grammar `.gr` files. **Code files will be submitted to Gradescope.** (You can also connect Gradescope to a GitHub repo.) Within each section there are conceptual questions. Answer these in a PDF file named `README.pdf`. **You will also upload `README.pdf` to Gradescope**. By the end of the homework you will have these additional files:

- `README.pdf`
- `grammar2.gr`
- `grammar3.gr`
- `grammar4.gr`
- `grammar_ec.gr` [extra credit]
- `randsent_ec.py` [extra credit]

Check Piazza for homework updates and to ask questions.

*Thanks to Alexandra DeLucia for editing an earlier version of this handout and improving its organization.

1 Random Sentence Generator

This is the coding section. You will need this script to work for the rest of the homework, so **finish this section early and feel free to ask for help**. Complete the `Grammar` class in `randsent.py` to generate random sentences given the following command-line arguments (command-line processing and the `main()` method are already implemented for you):

```
--grammar, -g
    Path to grammar file
--start_symbol, -s
    Start symbol of the grammar (default is ROOT)
--num_sentences, -n
    Number of sentences to generate (default is 1)
--max_expansions, -M
    Max number of nonterminals to expand when generating a sentence
--tree, -t
    Print the derivation tree for each generated sentence
```

Example usage generating two random sentences:

```
$ python3 randsent.py -g grammar.gr -n 2
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 ?
```

1.1 Reading and Storing the Grammar

The `grammar.gr` file provides a **probabilistic context-free grammar (PCFG)** like this one:

```
# 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
```

The above fragment provides these **context-free rules**:

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

Each line specifies one rule and consists of three **tab-separated** parts:

- a number (the relative odds of this sequence occurring; this is for sampling and is used in Section 2)
- 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; symbols are separated by whitespace.

You'll probably want to use a Python dictionary to store the rules. When reading the grammar, ignore comments (beginning with "#"), blank lines, and excess whitespace. (Consequently, allow grammar symbols to contain any character except whitespace, parentheses, and the comment symbol "#".)

1.2 Generating Sentences

Your `sample()` method must sample a random sentence from the probability distribution that is defined by the grammar. The grammar's start symbol is called `ROOT`, because it will be the symbol at the root of the derivation tree. We recommend implementing depth-first expansion. 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, your code 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.

Your code must work with *any* grammar file that follows the correct format, no matter how many rules or symbols it contains. So the grammar might be very different from example grammar, `grammar.gr`. Your program shouldn't assume anything in advance about the rules pppor their weights.

To avoid infinite loops, your generator should only expand `--max_expansions` times. The default is 450; this limit is further explained below. When this limit is reached the generator should return "`...`". So in the example discussed at the end of Section 1.3 below, you would print: `Sally found ... ... .`

More on weights The number before a rule 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, nor an integer.

In `grammar.gr` it happens that all the weights are 1, so all ways of expanding `NP` are equally probable. But this will change in section 2. So it will be easiest if you write your code to read and store weights now.

More on expansion limits 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. Furthermore, if you change the number on the second line to `1`, then the probability of terminating is 1, but the average size of the generated tree is ∞. (Try it out? Prove it?)

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. (Python's default limit on stack depth 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.

1.3 Printing Trees

In Section 3 your program will use the `--tree` option to print the sentences in tree form. Instead of

```
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 $\rightarrow$ floor and Noun $\rightarrow$ Adj Noun, among others.

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.

While it's not too hard to print the pretty indented format above, you don't have to worry about it. Your program only needs to generate a version without indentation,

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

and then "pretty printing" the tree is handled in `main()` by piping this linear tree through `prettyprint`, which adjusts the whitespace. That way, `randsent` can focus on the core NLP task, and `prettyprint` can focus on formatting—a "separation of concerns" between the two programs.

Once your program passes the maximum number of expansions, it will print "`...`" as before. For example, with a low limit of $M = 6$, you may get incomplete output like

```
(ROOT (S (NP Sally) (VP (VP (V found) ...) ...)) .)
```

The full tree would have been something like

```
(ROOT (S (NP Sally) (VP (VP (V found) (NP (Det the) (N piano)))
                        (PP (P in) (NP (Det the) (N garden))))) .)
```

Note that we printed only 6 nonterminals since $M = 6$. The green NP constituent would normally be printed by the 7th call (and its recursive calls), while the green PP constituent would normally be printed by the call after that. But since we have passed the limit of 6 at that point, those calls print `...` instead.

Suggestion (optional): You may eventually want to come back here and implement an alternative to the `--tree` option 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 later in the homework.

1.4 Questions

1. Provide 10 random sentences generated from your script.
2. Provide 2 random sentences generated from your script, using `--tree` to show their derivations.
3. As in the previous question, but with a `--max_expansions` of `5`.

2 Understanding Grammar Rules and Weights

Now that your `randsent.py` script is completed, we move onto the fun part: grammars!

2.1 Questions

1. Why does your program generate so many long sentences? Specifically, what grammar rule (or rules) is (or are) responsible and why? What is special about it/them?
2. The grammar allows multiple adjectives, as in `the fine perplexed pickle`. Why do your program's sentences do this so rarely? (Give a simple mathematical argument.)
3. Which numbers must you modify to fix the problems in item 1 and item 2, making the sentences shorter and the adjectives more frequent?

   Put these adjustments in a new grammar file named `grammar2.gr`. Check your answer by running your generator!
4. What other numeric adjustments can you make to `grammar2.gr` in order to favor more natural sets of sentences? Experiment. Explain the changes.
5. Provide 10 random sentences generated with the `grammar2.gr`.

2.2 Grammar Modification

Copy `grammar2.gr` to `grammar3.gr` and 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. Note that you'll need to add some words to the grammar.

1. `Sally ate a sandwich .`
2. `Sally and the president wanted and ate a sandwich .`
3. `the president sighed .`
4. `the president thought that a sandwich sighed .`
5. `it perplexed the president that a sandwich ate Sally .`
6. `that a sandwich ate Sally perplexed the president .`

   Note: Yes, 6 is acceptable in standard written English. It means the same thing as 5. Are there any other verbs that could replace `perplexed` in 5 and/or 6?
7. `the very very very perplexed president ate a sandwich .`
8. `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 4 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 mental grammar . . . so it shouldn't be grammatical according to your formal grammar either. The symbol * is traditionally used to mark unacceptable utterances.[1]

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.[2]

An important part of the problem is to *generalize* from the sentences above. For example, 2 is an invitation to think through the ways that conjunctions ("and", "or") can be used in English. 8 is an invitation to think about prepositional phrases ("on the desk," "over the rainbow", "of the United States") and how they can be used.

Code quality of your grammar The grammar file allows comments and whitespace because the grammar is really a kind of specialized programming language for describing sentences. Throughout this homework, 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.

Testing sections of your grammar Normally `randsent.py` generates entire sentences starting at `ROOT`. But if you just want to see what noun phrases look like, use the argument `-s NP` to start with the symbol `NP` and expand it fully.

2.3 Questions continued

9. Briefly discuss your modifications to the grammar.

10. Provide 10 random sentences generated with `grammar3.gr` that illustrate your modifications.

[1] 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 homework. Your criterion for "acceptable English" 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.

[2] 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.

3 Sentence Ambiguity and Parsing

In Section 1.3 you implemented the `--tree` option in `randsent.py` to print generated sentences in tree form. You will use it in this section's questions, along with the `parse` program we provided.

3.1 Questions

1. 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? Consider meaning.

3.2 Parsers

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.py` have generated the given sentence, and how?

Parsers are more complicated than generators. You'll write your own parser later in the course. For now, just use the `parse` script in the starter code.[3] Look at the top of the script for documentation. (A far faster version if you need it is available on the `ugrad` machines.[4]) Example usage:

```
./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.

[3] This script is written in Perl; you may need to install Perl, especially if you're using Windows.

[4] As `/usr/local/data/cs465/hw-grammar/dynaparse`. It uses a more efficient parsing algorithm, and it's a pretty tight implementation in compiled C++. Otherwise, it behaves almost identically to `parse`.

```
python3 randsent.py -g grammar.gr -n 5 | ./parse -g grammar.gr | ./prettyprint
```

Fun, huh?[5]

Use the parser to check your modifications to `grammar3.gr` to represent the phenomena in Section 2.2. Like `randsent.py`, the parser supports the `-s` option to specify a start symbol other than `ROOT`. Thus, if you just want to check that you can correctly parse a certain noun phrase with your grammar, you could run `parse` with the option `-s NP`.

If you did a good job on your grammar, then `./parse -g grammar3.gr` should be able to parse the example sentences from Section 2.2 *as well as similar sentences*. This kind of check will come in handy again when you extend your grammar in Section 4.

3.3 Questions continued

2. Use `python3 randsent.py -n 5 -t` to generate some random sentences from `grammar2.gr` or `grammar3.gr`, showing their derivations. Then try parsing those same sentences with the same grammar. (But don't try to parse a partial sentence that contains "`...`" due to `--max_expansions`.)

   Does the parser always recover the original derivation that was "intended" by `randsent`? Or does it ever "misunderstand" by finding an alternative derivation instead? Give a few examples and discuss.

3. 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).

4. 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? Give a few examples and discuss, then try the same exercise with `grammar3.gr`.

5. 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).

   (a) The first sentence reports

[5] Here's a tutorial on pipes and redirection. To get really fluent at manipulating text files at the command line, try Ken Church's Unix for Poets. And you may want to find more general tutorials on Linux and the Linux shell (i.e., command line). The classic book that I originaly learned from was Kernighan & Pike's *The Unix Programming Environment*, although it predates `bash`, which is the most popular shell today.

```
# p(best_parse)= 5.144e-05
# p(sentence)= 5.144e-05
# p(best_parse | sentence)= 1
```

- Why is `p(best_parse)` so small? What probabilities were multiplied together to get its value of `5.144e-05`? (*Hint:* Look at `grammar.gr`.)
- `p(sentence)` is the probability that `randsent` would generate this sentence. Why is it equal to `p(best_parse)`?
- Why is the third number `1`?

(b) The second sentence reports

```
# p(best_parse)= 6.202e-10
# p(sentence)= 1.240e-09
# p(best_parse | sentence)= 0.5
```

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`.)

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

```
# cross-entropy = 2.435 bits = -(-43.833 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.

(d) 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.)

(e) 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]
```

6. 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*?

   (a) 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.

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

   (c) Try to compute the entropy of the original `grammar`; what goes wrong and why?

7. 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.

4 Extending the Grammar

Now comes the main question of the homework! Think about all of the following phenomena, and extend `grammar3.gr` to handle ANY TWO of them—your choice. Briefly discuss your solutions and provide example output. **Save the new grammar in** `grammar4.gr`**.**

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 Section 2.2, **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 Section 2.2.

(c) *Relative clauses.* Examples:

- `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 Section 2.2. *Hint:* These sentences have something in common with (d).

(d) *WH-word questions.* If you also did (b), 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 (b), 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 (c).

(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 (a), 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

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

although if you also did (c), 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`.

4.1 Questions

1. Identify which two phenomena you chose to implement and describe how the list of changes you made to your grammar handles them.

5 Extra Credit: Extending Further!

Impress us! How much more of English can you describe in your grammar? Extend `grammar4.gr` in some interesting way (or create a wholly new grammar). For ideas, you might look at some random sentences from a magazine. Name the grammar file `grammar_ec.gr`.

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

5.1 Questions

1. Describe your additions.

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., https://www.madtakes.com/) 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.