Homework 8

For this homework, you should submit two files: `hw8.py` (questions 1–8) and `g2.cfg` (question 9).

1. We will create a grammar development tool. Define a class called `GDev`. The `__init__` method should take a name (a string) as input, and store it in the member `name`. For example:

```
>>> gd = GDev('g1')
>>> gd.name
'g1'
```

2. Define a method called `load_grammar`. It takes no arguments. It expects the file *name*`.cfg` to exist, where *name* is the GDev name. It loads a grammar from the file and stores it in the member `grammar`. For example, suppose that `g1.cfg` is as in Handout 15 #1.a.

```
>>> gd.load_grammar()
>>> gd.grammar
<Grammar with 26 productions>
```

3. Define a method called `reload`. It should call the method `load_grammar`, even if the grammar has already been loaded before. Then it should create a chart parser from the loaded grammar, and store the parser in the member `parser`. For example:

```
>>> gd.reload()
>>> gd.parser
<nltk.parse.chart.ChartParser object at 0x4121eb8c>
>>> gd.parser.grammar() is gd.grammar
True
```

4. Define a method called `parse`. It should take one argument, a string. It should call `word_tokenize` on the sentence, and pass the result to the parser. The `parse` method should return a single tree. If the parser returns more than one tree, then `parse` should return just the first one. If the parser does not return any trees, then `parse` should return `None`.

```
>>> print(gd.parse('Mary walked the dog'))
(S (NP (Name Mary)) (VP (V walked) (NP (Det the) (N dog))))
```

It is possible that the parser will signal an error: for example, if there is a word in the sentence that it does not know. In that case, we want `parse` just to return `None`. That is, `parse` needs to **trap** the error. Here is how to trap an error:

```
try:
    (call the parser here, return the first tree)
except:
    return None
```

The `except` block will only get executed if an error occurs. In this case, the value `None` will be returned if the parser signals an error.

```
>>> print(gd.parse('Gorp glimped'))
None
```

5. Define a method called `load_sents`. It takes no arguments. It expects the file *name*`.sents` to exist. The file should contain one sentence per line. Each sentence is either **good** or **bad**—good sentences are ones that the grammar ought to generate, and bad sentences are ones that the grammar should *not* generate. If the first character on the line is `'*'`, the sentence is bad, and otherwise it is good. The `load_sents` method should produce a list of pairs (*good, s*) where *good* is `True` for good sentences and `False` for bad ones, and *s* is the sentence itself (not including the `'*'`). The list of pairs should be stored in the member `sents`. Create a file `g1.sents` containing the sentences *Bob warbled, the dog ate my telescope,* and **Bob cat.*

```
>>> gd.load_sents()
>>> gd.sents[2]
(False, 'Bob cat')
```

You should also revise `reload` so that it calls `load_sents` in addition to (re)loading the grammar and creating the parser.

6. Define a method called `parses`. It should take no arguments. It should iterate through the pairs (*g, s*) in `sents`, and it should call `parse` on each sentence *s* in turn. For each sentence, it should print an empty line, then the sentence, then the result of calling `parse`. For example:

```
>>> gd.parses()

Bob warbled
None

the dog ate my telescope
(S (NP (Det the) (N dog)) (VP (V ate) (NP (Det my) (N telescope))))

Bob cat
None
```

7. Write a method called `regress` that takes no arguments. It should go through the pairs (*good, s*) in `sents`. For each, it should call `parse` on *s*. Define the *prediction* to be `True` if `parse` returns a tree, and `False` otherwise. If the prediction equals *good,* then the prediction is correct, and otherwise the prediction is wrong. For each pair, print out one line of output. The output line should start with `'!!'` if the prediction is wrong and `'  '` (two spaces) if it is correct. Then print out a space. Then print `'*'` if `good` is `False`, and a space if `good` is `True`. The output line ends with the sentence *s*. For example:

```
>>> gd.regress()
!!  Bob warbled
    the dog ate my telescope
   *Bob cat
```

8. Finally, the `__call__` method should simply call `reload` and `regress`. The idea is to use the set of example sentences to drive grammar development. One adds sentences, calls `gd()` to see which ones are being handled correctly or not, and then one edits the grammar to fix the prediction errors. After each file edit, one needs merely call `gd()` to see the revised grammar's predictions on the sentences. (Making sure that new revisions do not break things that previously worked correctly is known as **regression testing**.)

9. Write a new grammar called `g2.cfg` to cover the following sentences. Use the Handout 14 grammar as a guide, but it is probably best to start with a small grammar and gradually add to it. Do not try to handle the sentences all at once, but deal with them one at a time, revising your grammar to cover each one. That is, add them to your list of test sentences one at a time, and as you modify the grammar, run your development tool to make sure that you not only get the right parse for the new sentence, but you also don't break any of the old sentences.

The main goal is to parse the sentences correctly. You do not want the grammar to overgenerate profligately, but it is OK if the grammar allows some bad sentences. For example, it is OK if the grammar accepts "the dog have sleeping," as long as it assigns the correct parse to "the dog is sleeping." We will worry about tightening up the grammar later.

a. the dog is sleeping
b. I think that the cat barked
c. did the cat bark
d. the big red dog barked
e. the cat that barked chased Fido

Submit your grammar file `g2.cfg`.