Homework 8

The assignment is to implement a probabilistic version of the CKY parsing algorithm. Adding probabilities will actually simplify things a little, because there will only be one tree recorded in the chart, not a list of trees. I have provided the template `hw8.py` with some material already filled in for you, to get you started.

Background

The grammar `g2n.pcfg` is a modified version of the PCFG discussed in Handout 20; it is in Chomsky-normal form. To load a PCFG, read in the file as a string and use `PCFG.fromstring()`:

```
>>> from nltk import PCFG
>>> g = PCFG.fromstring(open('g2n.pcfg').read())
```

A PCFG is just like a CFG, except that the rules have two extra methods: `prob()` and `logprob()`. For example:

```
>>> from nltk import Nonterminal as NT
>>> r = g.productions(lhs=NT('Det'))[0]
>>> r
Det -> 'the' [0.6]
>>> r.prob()
0.6
>>> r.logprob()
-0.7369655941662062
```

You will also need to know how to create a tree with probabilities. There are only a couple differences from non-probabilistic trees: the class name is `ProbabilisticTree`, the constructor takes an extra keyword argument `logprob`, and there is a method `logprob()`. For example:

```
>>> from nltk import ProbabilisticTree as Tree
>>> t = Tree(r.lhs(), r.rhs(), logprob=r.logprob())
>>> t
ProbabilisticTree(Det, ['the']) (p=0.6)
>>> t.logprob()
-0.7369655941662062
```

The Chart

The first order of business is to implement the chart. The class `Node` represents entries in the chart; it is provided for you. Note that a node has four members: a nonterminal category `cat`, a start position `i`, an end position `j`, and a `tree`. It prints out in a convenient way.

```
>>> node = Node(r.lhs(), 0, 1)
>>> node.cat
Det
>>> node.i
0
>>> node.j
1
>>> node.tree
>>> node
Det(0,1)
```

The `Chart` class methods `__init__()` and `reset()` are also provided for you. A chart contains two dicts, `xijtab` and `jtab`. The `xijtab` is used to determine whether a node exists with a given category, start position, and end position. A key is a tuple `(cat, i, j)`, and the value is a node. The `jtab` is used to get the list of nodes ending at a given position. A key is an end position, and the value is the list of nodes ending at that position. Two other methods are also provided for you:

- The method `get()` returns a node, if it exists, and `None` if not.
- The method `reset()` clears the two dicts.

```
>>> chart = Chart()
>>> chart.xijtab
{}
>>> chart.xijtab[NT('Det'), 0, 1] = node
>>> chart.get(NT('Det'), 0, 1)
Det(0,1)
>>> chart.reset()
>>> chart.xijtab
{}
>>> chart.get(NT('Det'), 0, 1)
>>>
```

1. Implement the method `intern()`. It takes three arguments, `cat`, `i`, `j`, and returns the corresponding node. If no such node exists, it should create it and store it in `xijtab` and `jtab`. Storing it in `jtab` means adding it to the end of the list `jtab[j]`, creating the list if it does not already exist. For example:

```
>>> node = chart.intern(NT('Det'), 0, 1)
>>> node
Det(0,1)
>>> chart.intern(NT('Det'), 0, 1) is node
True
>>> chart.intern(NT('V'), 0, 1)
V(0,1)
```

```
>>> chart.intern(NT('VP'), 1, 3)
VP(1,3)
>>> chart.xijtab
{(Det, 0, 1): Det(0,1), (V, 0, 1): V(0,1), (VP, 1, 3): VP(1,3)}
>>> chart.jtab
{1: [Det(0,1), V(0,1)], 3: [VP(1,3)]}
```

2. Implement the method `ending_at()`. It takes one argument: an end position `j`, and returns the list of nodes that end at `j`. If there are none, it should return an empty list.

```
>>> chart.ending_at(1)
[Det(0,1), V(0,1)]
>>> chart.ending_at(2)
[]
```

The Parser

The skeleton of the class `Parser` is provided, including the `__init__()` and `reset()` methods. The `__init__()` method takes a grammar, and the `reset()` method takes a sentence (list of strings) as input. The parser has five members: `grammar`, `chart`, `words`, `new_nodes`, and `trace`. The sentence is stored in `words`, nodes that need to be processed are stored in `new_nodes`, and `trace` controls whether tracing information is printed out.

```
>>> parser = Parser(g)
>>> parser.grammar is g
True
>>> parser.chart
<hw8.Chart object at 0x...>
>>> parser.words
>>> parser.chart.intern(NT('NP'), 0, 1)
NP(0,1)
>>> parser.reset('the cat'.split())
>>> parser.words
['the', 'cat']
>>> parser.chart.xijtab
{}
>>> parser.new_nodes
[]
```

3. Implement the method `create_node()`. It takes four arguments: `rule`, `children`, `i`, and `j`. It should do the following.

- Let `cat` be the rule's lhs, and intern `(cat, i, j)` in the chart. Let `node` be the result. If `node.tree` is `None`, this is a *new node,* and otherwise it is old.

- If this is a new node, append it to the parser's `new_nodes` list.
- Compute `new_logprob` as follows. If `children` is a list containing a single string, then `new_logprob` is just the rule's logprob. Otherwise, the `children` are subtrees, and `new_logprob` is the sum of the rule's logprob and each of the subtree's logprobs.
- If this is a new node, or if `new_logprob` is greater than the logprob of `node.tree`, then create a new tree from `cat` and `children`, with logprob equal to `new_logprob`. Store the new tree in `node.tree`.
- If `trace` is `True`, then print the node, preceded by "`new`" or "`old`" depending on whether it is a new node or not.

For example:

```
>>> parser.trace = True
>>> parser.create_node(r, ['the'], 0, 1)
new Det(0,1)
>>> parser.chart.xijtab
{(Det, 0, 1): Det(0,1)}
>>> parser.new_nodes
[Det(0,1)]
>>> parser.new_nodes[0].tree
ProbabilisticTree(Det, ['the']) (p=0.6)
```

4. The method `shift()` takes one argument, an end position `j`. There is no return value. It should access the word w that ends at position `j`. As indicated in Handout 16, the first word in the sentence is considered to span positions 0 to 1; the second word spans positions 1 to 2; and so on. The `shift()` method should look for grammar rules of form $X \to w$, and for each such rule `r`, it should call

```
create_node(r, [w], j-1, j)
```

For example:

```
>>> parser.reset('the cat'.split())
>>> parser.shift(1)
new Det(0,1)
>>> parser.new_nodes[0].tree
ProbabilisticTree(Det, ['the']) (p=0.6)
```

5. The method `extend_edges()` takes one argument: `node`. It should look for previous nodes `p` that end where `node` begins. For each `p` that it finds, it should determine whether there is a rule `r` whose righthand side matches the categories of `p` and `node`. If so, call:

```
create_node(r, [p.tree, node.tree], p.i, node.j)
```

For example:

```
>>> parser.shift(2)
new N(1,2)
>>> parser.new_nodes
[Det(0,1), N(1,2)]
>>> node = parser.new_nodes[1]
>>> parser.extend_edges(node)
new NP(0,2)
>>> print(parser.new_nodes[-1].tree)
(NP (Det the) (N cat)) (p=0.18)
```

Note that the tree probability is the rule probability (0.6) times the probability for the Det subtree (0.6) times the probability for the N subtree (0.5).

6. The method `choose_node()` should choose a node from `new_nodes` to be processed. All the nodes in `new_nodes` will have the same end position, but they may vary in start position. `choose_node()` should pick a node with the *latest* available start position, remove it from `new_nodes`, and return it. One way to do that is as follows. Set `i` to 0. Iterate through indices `k` from 1 to the end of `new_nodes`. If the `k`-th node's start position is later than the `i`-th node's start position, set `i` equal to `k`. At the end, remove the `i`-th node from the list and return it. Tip: you can delete the `i`-th node from a list `lst` by doing:

```
>>> del lst[i]
```

Here is an example:

```
>>> parser.new_nodes
[Det(0,1), N(1,2), NP(0,2)]
>>> parser.choose_node()
N(1,2)
>>> parser.new_nodes
[Det(0,1), NP(0,2)]
```

(Note: this particular `new_nodes` list will never arise when the parser is actually running. In a real run, all nodes on the list will have the same end position.)

7. The method `run()` should do the following. Use a local variable `ptr` to point to the current word position, starting at 0. Then do the following repeatedly. If `new_nodes` is not empty, call `choose_node()` to get a node and pass it to `extend_edges()`. Otherwise, if `ptr` is at the end of the `words` list, break the loop. Otherwise, increment `ptr` and call `shift(ptr)`. Tip: you can loop forever by doing:

```
while True:
    if ready_to_quit():
        break
    ...
```

Here is an example:

```
>>> parser.reset('Mary walked the cat in the park'.split())
>>> parser.run()
new NP(0,1)
new V(1,2)
new Det(2,3)
new N(3,4)
new NP(2,4)
new VP(1,4)
new S(0,4)
new P(4,5)
new Det(5,6)
new N(6,7)
new NP(5,7)
new PP(4,7)
new NP(2,7)
new VP(1,7)
old VP(1,7)
new S(0,7)
```

8. Finally, the method `__call__()` should take a sentence (list of strings) as input and call `reset()` and `run()`. It should then see if there is a node whose category is the start symbol, spanning the entire sentence. If so, the return value is that node's tree. Otherwise, the return value is `None`.

```
>>> parser.trace = False
>>> t = parser('Mary walked the cat in the park'.split())
>>> type(t)
<class 'nltk.tree.ProbabilisticTree'>
>>> print(t)
(S
  (NP Mary)
  (VP
    (VP (V walked) (NP (Det the) (N cat)))
    (PP (P in) (NP (Det the) (N park))))) (p=0.00162)
```