Homework 9

The assignment is to implement the bottom-up chart-parsing algorithm of Handout 16, p. 5. To save you some work, the file `chart.py` contains an implementation of the chart data structure, and the file `g16.cfg` contains the grammar of Handout 16 #4.

The `chart` module

Creating and resetting chart. The `Chart` constructor takes no arguments:

```
>>> from chart import Chart
>>> chart = Chart()
```

One can turn on tracing by setting the `trace` member:

```
>>> chart.trace = True
```

When tracing is on, the chart prints out a message every time it creates a node, adds an expansion to a node, or creates an edge. One turns tracing back off by setting `chart.trace` to `False`.

One can simply do `print(chart)` to see all of the nodes and edges in the chart.

After processing a sentence, one can reset the chart to clear all the contents:

```
>>> chart.reset()
```

Creating a node. One can create a node in the chart:

```
>>> from nltk import Nonterminal as NT
>>> np = chart.create_node(0, NT(’NP’), 1, ’I’)
```

The arguments are: start position, category, end position, and expansion. The category must be a `Nonterminal`. The expansion may be either a string or an `Edge`. A new `Node` is created in the chart, with the given expansion, and the node is returned. However, if the node already exists in the chart, the expansion is added to it, but the return value is `None`.

Getting a node’s category. A node’s category is stored in the `cat` member:

```
>>> np.cat
NP
```

Fetching a node. One can test for the presence of a node in the chart, without changing anything, by using `get_node`:

```
>>> chart.get_node(0, NT(’NP’), 1)
Node(0, NP, 1)
>>> chart.get_node(0, NT(’S’), 2)
>>>
```

Creating an edge. One can also create edges in the chart.

```
>>> from nltk import CFG
>>> g = CFG.fromstring(open(’g16.cfg’).read())
>>> rules = g.productions()
>>> e1 = chart.create_edge(rules[0], np)
>>> e1
Edge(0 S -> NP *1 VP )
```

The two arguments to `create_edge` are the rule and the first child. The return value is an `Edge` object.

Edge properties. One can get the rule that an edge is working through, and the edge’s start and end positions:

```
>>> e1.rule
S -> NP VP
>>> e1.i
0
>>> e1.j
1
```

Note that the end position is actually the position of the dot. One can test whether the dot is at the end of the rule:

```
>>> e1.dot_at_end()
False
```

One can get the category after the dot:

```
>>> e1.after_dot()
VP
```

The method `after_dot` returns `None` if the dot is at the end.

Finding existing edges. One can search for edges by the sentence position at which they end (the position of the dot). The method is `get_edges_at`:

```
>>> for e in chart.get_edges_at(1):
...     print(e)
...
Edge(0 S -> NP *1 VP )
```

Extending an edge. To combine an edge with an additional child, one also uses `create_edge`:

```
>>> vp = chart.create_node(1, NT(’VP’), 2, ’walked’)
>>> e2 = chart.create_edge(e1, vp)
>>> e2
Edge(0 S -> NP VP *2 )
>>> e2.dot_at_end()
True
```

Creating a nonterminal node. A node's expansion may be an edge that has the dot at the end:

```
>>> s = chart.create_node(0, NT('S'), 2, e2)
```

Unwinding a node. One can get an iteration over the trees that a node represents by calling the node's `unwind` method:

```
>>> for t in s.unwind():
...     print(t)
...
(S (NP I) (VP walked))
```

Assignment

Note that the instructions here differ from the handout on some points; the reason for the changes is to make implementation and debugging easier. Most of the methods return no value. If the question does not explicitly say what the return value should be, assume there is none.

1. Define a class `Parser`. The constructor should take a filename as argument, and do the following. Load a CFG from the file and store it in the member `grammar`. Create a chart and store it in the member `chart`.

   Also define the method `reset`. It takes a sentence (a list of tokens) as input, and stores it in the member `words`. It also resets the chart, and sets the value of the member `todo` to the empty list.

   ```
   >>> parser = Parser('g16.cfg')
   >>> parser.reset(['foo', 'bar'])
   >>> parser.words
   ['foo', 'bar']
   >>> parser.todo
   []
   ```

2. Implement `create_node` and `create_edge`. They take the same arguments as the corresponding `Chart` methods. Each should simply call the corresponding `Chart` method, and append the resulting node or edge to the `todo` list. (However, do not append `None` to the `todo` list.)

   These instructions differ from the handout. The goal is to make the parser methods independent of each other, so that you can test them individually. The `todo` list will be processed by the method `next_task` (question 7).

3. Implement the `shift` method. Contrary to the handout, its argument should be the index of the word to shift, i, and it should create nodes spanning positions i to $i+1$. Create one node for each part of speech that the grammar assigns to `words[`i`]`. You may assume that the members `grammar`, `chart`, and `words` are all appropriately set.

```
>>> parser.chart.trace = True
>>> parser.reset([’book’])
>>> parser.shift(0)
Chart: created Node(0 N 1) with expansion ’book’
Chart: created Node(0 V 1) with expansion ’book’
>>> parser.todo
[Node(0 N 1), Node(0 V 1)]
```

4. Implement the `bu_predict` method. It takes a `Node` as input. Let X be the node’s category. For each rule r whose righthand side begins with X, call `create_edge` on r and the node.

```
>>> parser.reset([’I’, ’book’, ’a’, ’flight’, ’in’, ’May’])
>>> parser.shift(0)
Chart: created Node(0 NP 1) with expansion ’I’
>>> node = parser.todo.pop()
>>> parser.bu_predict(node)
Chart: created Edge(0 S -> NP *1 VP)
Chart: created Edge(0 NP -> NP *1 PP)
>>> parser.todo
[Edge(0 S -> NP *1 VP), Edge(0 NP -> NP *1 PP)]
```

Note: the list method `pop` removes the last element from the list, and returns it. It essentially undoes the last `append`.

5. Implement the `extend_edges` method. It takes a `Node` as input. It iterates through the edges e that end where the node begins, and if the node’s category is the same as the category after the dot in e, then a new edge is created that combines e and the node. (Use `create_edge`.)

```
>>> vp = parser.chart.create_node(1, NT(’VP’), 2, ’fake’)
Chart: created Node(1 VP 2) with expansion ’fake’
>>> parser.extend_edges(vp)
Chart: created Edge(0 S -> NP VP *2 )
```

6. Implement the `complete` method. It takes an `Edge` as input. For safety, it should signal an error if the dot is not at the end. Create a node corresponding to the lefthand side of the rule, with the edge as its expansion, covering the same span as the edge.

```
>>> e = next(parser.chart.get_edges_at(2))
>>> e
Edge(0 S -> NP VP *2 )
>>> parser.complete(e)
Chart: created Node(0 S 2) with expansion Edge(0 S -> NP VP *2 )
```

7. The method `next_task` takes no input. It expects `todo` to be non-empty. It removes the last item from `todo` and processes it. If the item is a node, it calls bu_predict and extend_edges on it, and if the item is an edge, and the dot is at the end, it calls complete on it.

8. Implement the method fill_chart. It should call shift for each word, and after each time it calls shift, it should call next_task repeatedly until the todo list is empty.

```
>>> parser.reset(['I', 'book', 'a', 'flight', 'in', 'May'])
>>> parser.fill_chart()
Chart: created Node(0 NP 1) with expansion 'I'
Chart: created Edge(0 S -> NP *1 VP)
...
Chart: created Edge(1 VP -> V NP *6 )
Chart: added expansion Edge(1 VP -> V NP *6 ) to Node(1 VP 6)
```

9. Make the parser callable. When it is called as a function, it should take a sentence (that is, a list of tokens) as input. It should call reset and fill_chart. Then, if there is a node that spans the whole sentence and whose category is the grammar's start symbol, it should return an iteration over the trees of that node. Otherwise, it should return an empty iteration. Note: if a method calls yield for some inputs but not others, the result will be an empty iteration for the inputs where it never calls yield.

```
>>> parser.chart.trace = False
>>> for t in parser('I book a flight in May'.split()):
...     print(t)
...
(S
  (NP I)
  (VP (VP (V book) (NP (Det a) (N flight))) (PP (P in) (NP May))))
(S
  (NP I)
  (VP (V book) (NP (NP (Det a) (N flight)) (PP (P in) (NP May)))))
```