CS159 - Assignment 4
Due: Friday, Mar. 11 at 5pm

In the last assignment we extracted PCFG rules using maximum likelihood estimation from an existing treebank and then binarized the grammar. In this assignment, we will now take a binarized grammar and use it to parse a new sentence using the CKY parsing algorithm.

For this assignment, I am again giving you a fair amount of flexibility for how you implement it. You may use whatever programming language you like as long as it is installed on the lab machines (which should be most of them you'd want to use). I have, however, provided you with some starter code again in Java for representing rules that will probably make your life easier.

You may (and I would encourage you to) work with a partner on this assignment. If you do, you must both be there when either of you are working on the project and you should only be coding on one computer (i.e. pair programming). If you would like a partner, but don't have one, e-mail me asap and I will try and pair you up.

1 CKY parsing

Create a CKY parser. Your parser should be initialized based on a binary grammar. Once it is initialized, it should be able to parse multiple sentences without having to reread in the grammar. Your parser must be efficient, which means you need to be efficient about many of the lookup operations in the algorithm.

To get you some very basic experience playing with command-line scripts (and because I'm allowing you to use any language you want), along with your code, you will also be submitting a script called `parse.sh` that allows you to run your parser from the command-line. Your script should take two command-line parameters as input. The first, should be the name of the grammar file and the second, the name of a file consisting of sentences to be parsed, with one sentence per line. Your script should then output the most-likely parse for each sentence along with the score for that parse separated by a tab, one per line. The parse tree output should be the parenthesized format we saw before for assignment 2. If the sentence does not have parse, just output NULL.

For example, the following very basic script would work for a java version:

```
#! /bin/bash

java -Xmx1G nlp.parser.CKYParser $1 $2
```

assuming that the Java parser took two parameters from the command-line. If this is confusing or you have troubles with this, please come talk to me sooner than later (don't wait until the last minute to ask me about this!).

When you think you have it all working, parse the file `test.senteces` with the large grammar file `full.pcfg`. My parser takes about 2 minutes to parse these 10 sentences (though one doesn't have a parse). If you're not careful about efficiency, yours can take much, much longer.

Grammar rules

For the last assignment, to keep debugging simpler, we learned PCFG rules. However, if you try and parse with PCFG rules you can have underflow problems as you start to multiply together small probabilities. Instead, we're going to use rules that have a score/weight associated with them. In our case, the weights will be the log of the probabilities, but in general, we could use any arbitrary weights. Unlike probabilities, with weights the score of a parse is now the **sum** of the weights of the associated rules, and we would still like to pick the parse tree with the largest sum. For log probs, these sums will be negative.

A rule is of the form:

```
LHS -> RHS1 RHS2		WEIGHT
```

where RHS2 does not exist if it's a unary rule and there are 2 tabs between the last RHS symbol and the weight.

The other change that I have made to the grammar is that all lexical rules (that is that have a lexical RHS) now have the lexical component start with '* '. For example:

```
DT -> * a		0.0
```

is the rule stating that a determiner (DT) goes to the lexical item 'a' with weight 0.0 (probability 1).

2 Provided Code/Data

I have provided you with some resources to get you started.

- **Code:** I have provided you with an updated rule class called `GrammarRule` that handles the updated rule format. In particular, it now uses weights instead of probabilities and also keeps track of whether or not the rule is a lexical rule or not (see the `isLexical` method).
- **Data:** I have provided you with a number of grammars and some sample output to get you started. `example.output` contains the parser output from parsing `example.input` using the

grammar file `example.pcfg`. `full.pcfg` contains a very large grammar that should be able to parse most sentences (though we're not dealing with out of vocabulary, so if the sentence has a new word, it won't parse).

How to Proceed

The following is how I would suggest proceedings, though you're welcome to do it however makes the most sense for you:

1. Make sure you understand the CKY algorithm. We walked through the algorithm in class and the book provides pseudo-code. What information are you going to need to store in your table? All of the table access operations need to be fast ($O(1)$). How are you going to store things in the table to make this work?

2. Work through at least one example using the `example.pcfg`, filling in the table on paper. Write down the scores, etc. and save later for testing.

3. Write the constructor for your parsing class which will read in the grammar rules and store them in an efficient manner. I suggest storing them in three separate data structures, one for lexical rules, one for non-lexical unary rules and one for binary rules. Think about how you're going to be accessing each of these in the algorithm.

4. Write a helper class representing one entry in your CKY table. What do you need to store? What types of questions will you need to support? How can you answer these questions efficiently? It may be simpler not to worry about storing the actual back references for reconstructing the parse and just worrying about the weight to start with.

5. Start writing your parsing method. Create a new CKY table (two dimensions) and fill in the diagonals based on the the lexical components. I would suggest writing a method that adds a constituent to your table (either in your entry class or as a standalone method). Print out the added constituents (either as they're added or by printing out the table) and make sure everything is added appropriately. You can compare against your hand-written example.

6. Add the check for unary rule applications in your add method. *Every* time you add a new constituent (whether it be from a binary or unary rule) you need to check to see if any unary rules apply. Make sure to avoid infinite loops by checking that by adding the constituent, you're getting a better parse. Also, make sure you've stored your unary rules in such a way that checking if any unary rule applies is *fast*. Check again against your hand-crafted test examples.

7. Write the main loops to fill in the table from top to bottom. If you're good about debugging your add method beforehand, you should just be able to focus on the loops and not on adding things to the table. You should be able to get a full parse now. Check your result against your hand example. Make sure the weights are right.

8. Work on actually reconstructing the parse. For each added constituent, you'll need to keep track of what subconstituents it came from. This may require creating a new data structure that's going to feel similar to a simplified version of `ParseTree`. You'll also probably need to modify your table entry class. Once you have the backpointers setup, try and print out the parse. Recursion will be your friend here. A parse tree is a recursive structure.

3 Hints

- If you need more memory (which you may) you can give the Java virtual machine more memory by adding `-Xmx1G` as a flag (which will set your heap at 1G). From the command line, just add it after `java`. From `Eclipse` you'll need to add it as a VM argument under `Run->Run Configurations...` and then under the `Arguments` menu.
- To get things to run efficiently, you'll need to use hash tables in a number of places.
- Debug your code incrementally and make sure to print things out as you go. If you wait until the end to try and debug it, you're going to have a hard time.
- We're only looking for the best parse, which means for any entry in the table, we only need to store the best version of each constituent.
- If you get stuck on printing out the parse, take a look at the `ParseTree` class and it's `toString` method from the last assignment. The idea should be fairly similar.
- Unary rules are tricky. Make sure that any time you add a new constituent, you add a unary rule, but be careful about not getting stuck in infinite loops.
- Spend some time making sure you understand the algorithm and design your code. It's not a ton of code to write, but it does take some thought figuring out how everything fits together.

4 Extra credit

There are two possibilities for extra credit on this assignment. Each is worth 3 points.

- **Real trees:** Right now, we're outputting binarized versions of the trees. We'd really like to get the original grammar trees back out. Add a flag `-original` to your `parse.sh` script that outputs original grammar trees. Make sure that the default behavior is still the binarized version. Include in your `output` folder a parsed version of `test.sentences` called `test.sentences.parsed.original` containing these new parses.
- **n-best list:** Rather than just finding the best parse tree, we might also want to output some fixed number of best parse trees. Add a flag `-nbest` <num> to your `parse.sh` script that outputs the <num> best parses for each input sentence. Include in your `output` folder an nbest parsed version of `test.sentences` called `test.sentences.parsed.nbest` containing the 10 best parses for each test sentence.

5 When you're done

When you're all done, follow the directions on the course web page for submitting your work in the dropbox. Make sure that your code compiles, that your files are named as specified. If you get an error when submitting, try changing the name of the folder to include a version number and resubmit.

To make things easier for me, include one sub-folder with your code called `code` and another sub-folder with your output called `output` in nested within your submitted directory.

If you worked with a partner, put both people's last names on the submitted directory, but only submit one copy.

What to submit

In your `code` subdirectory, you should submit your full working code. If the naming of the files isn't obvious about where various things are being done, please include a `README` file explaining your organization. In this file should also be included your `parse.sh` script for running your parser from the base directory (i.e. from within code).

In your `output` directory you should have the following a file call `test.sentences.parsed` representing your parse of `test.  sentences` using `full.pcfg`.

Please make sure to follow the folder outline and file naming conventions above.

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file you modify.
- Each class and method should have appropriate JavaDoc comments (doc strings if you do it in Python).
- If anything is complicated, put a short note in there to help me out if there are any issues.

This is a non-trivial assignment and it can get complicated, which makes code style and comments very important so that I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

Grading

Part	points
Parser	50
Efficiency	10
style/commenting	5
extra credit	6
total	65 + 6 extra