cs294-5: Statistical Natural Language Processing
Assignment 3: A Treebank Parser

Due: Nov 5th

Setup: The code for assignment 3 is under `/home/ff/cs294-5/java/src3`. The precompiled classes are in `home/ff/cs294-5/java/classes3`. The data for this assignment is in `/home/ff/cs294-5/corpora/assignment3`, and consists only of the parsed WSJ section of the Penn Treebank.

The starting file for this assignment is

```
…/assignments/PCFGParserTester.java
```

Make sure you can run the main method of the `PCFGParserTester` class. You can either run it from the data directory, or pass that directory in as the sole command line parameter to the class:

```
java –server –mx500m edu.berkeley.nlp.assignments.
                        PCFGParserTester /home/.../assignment3
```

Description: In this project, you will build a broad-coverage parser. The bulk of the support code assumes that you will build agenda-driven PCFG parser, but you're free to build a beam-decoded shift-reduce parser, or implement any other parsing solution you find interesting and manageable. I will list the data flow first, then describe the support classes that I've provided. You are free to use these classes, or not, as you see fit.

Currently, files 200 to 2199 of the Treebank are read in as training data, 2200 to 2299 are validation data, and 2300 to 2319 are test data (you can look in the data directory if you're curious about the native format of these files). You can run on fewer files to speed up your preliminary experiments. Using the training trees, a `BaselineParser` is constructed, implementing the `Parser` interface (which only has one method: `getBestParse`). The parser is then used to predict trees for the sentences in the test set. The static integer MAX_LENGTH determines the maximum length of sentences test on (it does not affect the training set). You can lower MAX_LENGTH for preliminary experiments, but your final parser should work on sentences of at least length 20 in a reasonable time (5 seconds per 20-word sentence should be achievable without too much optimization).

This baseline parser is really quite poor – it takes a sentence, tags each word with its most likely tag (i.e. a unigram tagger), then looks for occurrences of tag sequence in the training set. If it finds an exact match, it answers with the training parse of the matching training sentence. If no match is found, it constructs a right-branching tree, with nodes' labels chosen independently, conditioned only on the length of the span of a node. If this

sounds like strange (and terrible) way to parse, it should, and you're going to provide a better solution.

You should familiarize yourself with these basic classes:

`Tree`	CFG tree structures, (pretty-print with `Trees.PennTreeRenderer`)
`UnaryRule/BinaryRule/Grammar`	CFG rules and accessors

You will be using the former class no matter what kind of parser you build. I'm assuming most people will build agenda-driven PCFG parsers. In that case, you will also find `util.PriorityQueue` useful. It is a fast priority queue implementation, but it does not support a promotion / increasePriority operation. If you want to increase the priority of an edge in the queue, you need to add that edge a second time, with the new higher priority, and be aware that whenever you pop an edge off of the queue, it might be a duplicate "dead" edge. I made this design decision because, in my experience, highly-tuned agenda-driven parsers spend a large fraction of their time dealing with the overhead that queues supporting promotion incur. But you are of course welcome to change the priority queue if you'd really like it to support promotion. I also provided a basic lexicon (`Lexicon`) – this lexicon is minimal, but handles rare and unknown words adequately for the present purposes. If you want to employ your tagger from assignment 2 instead of using this lexicon, that is perfectly acceptable.

The first thing you should do is scan through a few of the training trees to get a sense of the format and range of inputs. Something you'll notice is that the grammar has relatively few non-terminal symbols (27 plus part-of-speech tags) but thousands of rules, many trinary-branching or longer. As we discussed in class, most parsers require grammars in which the rules are at most binary branching. You can binarize and unbinarize trees using the `TreeAnnotations` class. The implementation I gave you binarizes the trees in a way that doesn't generalize the grammar at all. You should run some trees through the binarization process to get the idea of what's going on. If you annotate/binarize the training trees, you should be able to construct a `Grammar` out of them using the constructor I provided. This grammar is composed of binary and unary rules, each bearing its relative frequency estimated probability from the training trees you provide it. It encodes a PCFG, and your goal is to build a parser which parses novel sentences using that PCFG (unless you leave the beaten path for this assignment). Building this parser is expected to be the bulk of the work for this assignment.

Once you've got a parser which, given a test sentence, returns a parse of that sentence using the grammar from the training trees, you have several choices in how to proceed with this assignment. One choice is to focus on the grammar, using annotation and binarization techniques like horizontal and vertical markovization to improve the accuracy of your parser. The current representation is equivalent to a 1st-order vertical process with an infinite-order horizontal process. Everyone should at least try out a 2nd-order / 2nd-order grammar, meaning using parent annotation (symbols like `NP^S` instead of `NP`) and forgetful binarization (symbols like `@VP->…NP_PP` which omit details of the

history, instead of `@VP->_VBD_RB_NP_PP` which record the entire history). If you choose to focus on this aspect, however, you should do more exploration of these kinds of issues.

Another choice is to focus on efficiency of the parser. You could compare a beam method with the exact agenda-driven method, or implement a pruning technique like an A* heuristic or a figure-of-merit and compare it to the basic agenda method (this may require substantial effort). Or you could study the trade-offs of pre-tagging the sentence before parsing, rather than letting the parser do the tagging.

A final choice is to investigate some other aspect of parsing that can be easily tested in your existing code. For example, you might investigate whether some compact subset of a grammar gives nearly the same accuracy. Or if you're interested in cognitive issues, you could study whether correct trees really do tend to have bounded stack depths in one direction or the other (and whether incorrect trees perhaps have different profiles). Or you could do an error analysis of the mistakes your parser makes.

To summarize my expectations in this assignment, I expect that the bulk of your time will go into understanding the setup and building a basic PCFG parser (probably agenda-driven). Once that is basically working, I expect you to be able to try out a 2^{nd}-order / 2^{nd}-order grammar with a very small amount of additional work; this grammar should work much better. You should also report some errors that you parser seems to make often, though you need not do a detailed data analysis. Then, I expect you to very briefly do some other kind of experiment that you find interesting. I do NOT expect this last part of the assignment to be as much work as building the parser in the first place, though of course it could if you wanted it to. Not all of the extensions I mentioned are possible unless you do want to spend substantial additional time coding.

Coding Tips: Whenever you run the java VM, you should invoke it with as much memory as you need, and in server mode:

```
java –server –mx500m package.ClassName
```

You'll get much faster performance than just running with no options.

If your parser is running very slowly, run the VM with the `–Xprof` command line option. This will result in a flat profile being output after your process completes. If you see that your program is spending a lot of time in hash map, hash code, or equals methods, you might be able to speed up your computation substantially by backing your sets, maps, and counters with `IdentityHashMap` instead of `HashMap`. This requires the use of something like an `Interner` for canonicalization… ask around if you're not sure what that would entail, some people in the class already know this trick.