cs294: Statistical Natural Language Processing
Assignment 5: Treebank Parsing

Due: April 16th

In this assignment, you will build an English treebank parser. You will consider both the problem of learning a grammar from a treebank and the problem of parsing with that grammar.

Setup: The data for this assignment is available on the web page as usual. It uses the same Penn Treebank data as the first and third assignments, this time with full parses.

The starting class for this assignment is

```
edu.berkeley.nlp.assignments.PCFGParserTester
```

Make sure you can access the source and data files.

Description: In this project, you will build a broad-coverage parser. You may either build an agenda-driven PCFG parser, or an array-based CKY parser. I will first list the data flow, then describe the support classes that are 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, as is standard for this data set. Depending on whether you run with `-validate` or `-test`, either files 2200 to 2299 are read in (validation), or 2300 to 2399 are read in (test). You can look in the data directory if you're curious about the native format of these files, but all I/O is taken care of by the provided code. You can always run on fewer training or test files to speed up your preliminary experiments, especially while debugging (where you might first want to train and test on the same, single file).

Next, the code constructs a `BaselineParser`, which implements the `Parser` interface (with only one method: `getBestParse()`). The parser is then used to predict trees for the sentences in the test set. You can control the maximum length of training and testing sentences with the parameters `-maxTrainLength` and `-maxTestLength`. The standard setup is to train on sentences of all lengths and test on sentences of length less than or equal to 40 words. 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; a good research parser will parse 20 word sentences in more like 0.1 seconds).

This baseline parser is quite terrible – it takes a sentence, tags each word with its most likely tag (i.e. runs a unigram tagger), then looks for that exact tag sequence in the training set. If it finds an exact match, it answers with a known parse for that tag sequence. 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 seems like a strange way to parse to you, it should. 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 these classes no matter what kind of parser you build. If you choose to build an agenda-based parser, you will find `util.GeneralPriorityQueue` useful. If you choose to build an array-based CKY parser, you will instead find the `UnaryClosure` class (also in the harness file) useful. It computes the reflexive, transitive closure of the unary subset of a grammar, and also maps closure rules to their best backing paths.

I have 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 to manually 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 (including yours) require grammars in which the rules are at most binary branching. You can binarize and unbinarize trees using the `TreeAnnotations` class. The default implementation binarizes the trees in a way that doesn't generalize the n-ary grammar at all (convince yourself of this). You should run some trees through the binarization process and look at the results to get an 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 provided. This grammar is composed of binary and unary rules, each bearing its relative frequency estimated probability from the training trees you provide on construction. It therefore encodes a PCFG, and your goal is to build a parser which parses novel sentences using that PCFG. Building this parser is the bulk of the work for this assignment.

Extensions: 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 better annotation 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 horizontal 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 various kinds of annotation.

Another choice is to focus on efficiency of the parser. You could, for example, compare a beam method with an exact parsing method, or implement a pruning technique like an A* heuristic or a figure-of-merit and compare it to an exhaustive approach.

Still another 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 the 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 a substantial 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 (agenda-driven or CKY). Once that is 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 at least some errors (with examples) that you parser seems to make often, though you need not do a detailed data analysis unless you want to focus on that. Finally, 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. 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 a `util.Interner` for canonicalization… ask around if you're not sure what that would entail, some people in the class already know this trick.