# CSE 517, Fall 2013: Assignment 3

**Due:** Saturday, Feb 23th at 5pm

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. The data and support code are available on the course Dropbox in Catalyst. Please submit two files: a PDF (with your name) containing your writeup and an archive (zip, tar.gz, etc.) including all the code.

The data is taken from the Penn Treebank and includes sentences paired with complete parses. The provided support code is in Java and we highly encourage, but do not require, you to use it. In either case, the code submitted must provide detailed documentation.

The starting class for this assignment is

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

## 1 Building a Parser (50%)

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. We will first go over 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, or even just a single tree). Be sure to only use the test set once for each model (twice, if you absolutely need it) and use the validation set for development.

Once the trees are read, 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 a 20 word sentence 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. This baseline is just a crazy placeholder - youre 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.

We have also provided a basic lexicon (`Lexicon`) to associate words with part-of-speech tags. This lexicon is minimal, but handles rare and unknown words adequately for the present purposes (see the javadoc for details). If you want to employ your tagger from Assignment 2 instead of using this lexicon, that is perfectly acceptable.

At this point, you should manually scan through a few of the training trees to get a sense of the format and range of inputs. Something youll 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 doesnt 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 whats 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 main goal is to build a parser which parses novel sentences using that PCFG. Building this parser will be the bulk of the work for this assignment.

## 2 Better Grammar (50%)

Once you have a parser which, given a test sentence, returns a parse of that sentence using the training grammar, you will focus on improving performance by modifying the grammar using better annotation/refinement techniques. For example, you may use 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. You 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 abstract the horizontal history, instead of `@VP->_VBD_RB_NP_PP` which record the entire history). In addition, you are required to explore further annotation methods. A good submission will cover all the tag splitting options we discussed in class and provide detailed evaluation of each method's contribution (for example, using ablation tests). More ideas for tag splitting can be found in Klein and Manning (2003). Alternatively, you may choose to lexicalize your grammar (see Michael Collins' notes on the course website). **A good and reasoned approach to lexicalization will receive a bonus of 30%**.

## 3 Writeup (max. 4 pages)

For the write-up, we want you to describe what you've built and analyze your results. Describe your grammar annotation / refinement pipeline and ablate the various steps to evaluate their contribution. Ablation tests should be done using the validation (development) set. Summarize your results in a table and discuss them. Back your annotation / refinement choices with reasoned explanations. Just claiming that they provide better performance is not enough, we want to see that you understand how your improvements relate to the actual problem. You should also report at least some errors (with examples) that your parser seems to make often. Try to discuss potential improvements that might overcome these errors. Finally, report your final evaluation results using (1) the initial grammar, (2) the 2nd-order / 2nd-order grammar and (3) the final grammar refinement you chose to use. All your results should be reported using the parser your built.

## Coding Tips

Whenever you run the java VM, you should invoke it with as much memory as you need (and in server mode for JVMs which dont default to sever):

```
java -server -mx500m package.ClassName
```

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 operations or `hashcode` / `equals` methods, you might be able to speed up your computation substantially by backing your sets, maps, and counters with `IdentityHashMaps` instead of `HashMaps`. This change requires the use of something like a `util.Interner` for canonicalization.

**The assignment was adapted from Dan Klein's CS 288 Course at UC Berkeley.**

## References

Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, pages 423–430, Stroudsburg, PA, USA. Association for Computational Linguistics.