# CSE 517: Assignment 3

**Due: at 5:00pm on November 22nd**

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%)

You will build an array-based CKY parser. We will first go over the data flow, then describe the support classes that are provided.

**Dataset:** 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. You can control the maximum length of training and testing sentences with the parameters -maxTrainLength and -maxTestLength. 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). The standard setup is to train on sentences of all lengths and test on sentences of length less than or equal to 40 words.

**Baseline:** The BaselineParser implements the Parser interface (with only one method: getBestParse()). The parser is then used to predict trees for the sentences in the test set. 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 - you're going to provide a better solution.

**Grammar:** 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 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.[1] 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 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.

**Relevant Classes:** You should familiarize yourself with these basic classes:

- Tree: CFG tree structures (pretty-print with Trees.PennTreeRenderer)
- UnaryRule/BinaryRule/Grammar : CFG rules and accessors
- 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 the lexicon, that is perfectly acceptable.

## 1.1 Implementation details

**Data Structures:** You will need to implement the dynamic programming data structures required for computing the maximum scoring parse under a PCFG. You need a structure which stores the maximum score for a sub-tree rooted by a symbol $X$, covering the $i$th to $j$th words in the sentence. You will also need structures that store a backpointer for each $(i, j, X)$ element that stores which rule was used (it should be of the form $X \to Y_1 Y_2$, in the case of binary rules, or $X \to Y_1$ in the case of unary rules) and the split of the sentence used by the rule (a number $k$ such that $k \in \{i..(j-1)\}$). One way to way to implement these structures that is fast enough is to use arrays indexed by ids for every symbol [2].

**Unary Rules:** In class, we assumed all rules are binary. Here, we ask you to extend those algorithms to support unary rules. Unary rules are a special case of the grammar. They take the form $X \to Y_1$. The key to parsing with unary rules is that they must not be cyclical. The code provides the UnaryClosure class to simplify processing the unary rules. When you create the UnaryClosure class with a Grammar, it computes the maximum score of the path from $X$ to $Y'$,

[1] You may want to confirm that you understand the importance of this from a computational point of view.

[2] Since we known that $i <= j$ at all times in the dynamic program, your implementation could take only $\frac{L(L-1)}{2}$ space, where $L$ is the length of the sentence, but it is ok if it takes $L^2$, because it simplifies the indexing and is fast enough for our purposes.

using only unary rules, and also stores the path that achieved that score. When you query the UnaryClosure class for rules headed by $X$, you get all pairs that start with $X$ and go to some $Y'$ and the score of the optimal path between $X$ and $Y'$. $X \to Y'$ might not be a rule in your original grammar, but the path backing that pair follows rules from your original grammar. Later, when you need to reconstruct the path between $X$ and $Y'$, you can again query the UnaryClosure class. In parsing, when computing the max for spans of length $n$, you must consider all $X$, $Y'$ pairs from the UnaryClosure before moving on to spans of size greater than $n$.

**Performance and Testing:** It may be difficult to confirm that your parser is working, especially on a first pass implementation that is not optimized. First, confirm your code is working on extremely small sentences ( $\leq 5$ ) and then move your way up. Parsing results should be better for shorter sentences than longer sentences because there is less ambiguity. For sentences less than 5 in length, expect performance in the mid 80's on F-measure; for less than 10 in length, performance in the low 80's and for less than 20 in length in the high 70's. For the full dev set, sentences less than 40, the initial grammar should be in the low 70's, but expect to wait a few hours to get these results (unless you've really worked on making your code fast). On the other hand, you can get results on sentences of length less than 10 in a few minutes, even with a not optimized implementation. A good strategy for improving efficiency is to get the parser working on the small sentences and then make modifications to the code that allow it run on longer sentences while confirming you get identical performance on shorter sentences.

### 1.2 Better Grammar (50%)

Once you have a parser which, given a 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 implement a 2nd-order / 2nd-order grammar, meaning using parent annotation (symbols like NPˆS 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 one other annotation method. A good submission will 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) or in the class notes. For extra credit, you may choose to lexicalize your grammar (see Michael Collins' notes on the course website). Warning: this can be very difficult. A good and reasoned approach to lexicalization will receive a bonus of 30%.

### 1.3 Write-up (max. 4 pages)

For the write-up, we want you to describe what you've built and analyze your results. We expect each of the following items in your report:

- Describe the parsing algorithm you implemented to enough detail that we can confirm you understand it. 3-5 sentences containing a sketch of the algorithm will be enough along with any details you used for making it run fast. Be sure to highlight how you handled unary rules.
- Give development set results for the initial grammar.

- Describe your grammar annotation / refinement pipeline. Back your annotation / refinement choices with reasoned explanations. Claiming that they improve performance is not enough, we want to see that you understand how your improvements relate to the actual problem.

- 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. One way to tabulate the results would be to put versions of the system in the rows and measures of performance in the columns. Including a reference, for example the full system using all the functionality you developed, is common practice.

- Report a few errors (with examples) that your parser makes often. Discuss potential improvements that might overcome these errors.

- Report your final test results using (1) the initial grammar, (2) the 2nd-order / 2nd-order grammar and (3) the final grammar refinement you chose to use.