

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.863J/9.611J, Natural Language Processing
Laboratory 2: Parsing Parsing

Handed out: February 27, 2009

Due: March 13, 2009

Goals of the Laboratory. Laboratory 2 is an introduction to context-free parsing: learning how ambiguity makes natural language parsing difficult, and learning a bit about two basic algorithmic approaches to solving this problem, one stack-based, and the other using a data structure called a “chart” to ‘memoize’ and re-use already-built structure. Re-using already-built structure is the key to efficient context-free parsing. We also introduce you to probabilistic context-free parsing: how this speeds up sorting through the tremendous number of parses that can arise even with simple sentences when using more than a “toy” grammar; how to infer rules from already-parsed examples sentences; and some of the strengths and weaknesses of this simple learning approach. In this lab, you’ll have a chance to work with an actual, large-scale corpus, the Penn Treebank, which will give you familiarity with the data conventions commonly referenced in computational linguistics and NLP papers. Finally, this laboratory *will* involve a little Python programming. If you feel even the tiniest bit insecure about this, and/or you need some assistance, *please* see either TA, and sooner rather than later. They’ll be glad to guide you.

After completing this laboratory, you should be able to understand and use both the classical and the modern probabilistic “tools of the trade” for context-free parsing, as well as some of their limitations.

What you must turn in. As before, please email your write-up as a pdf file to 6.863-graders@mit.edu. Like Laboratory 1, Part 1, most of the questions that follow require only short answers. You may use the write-up template provided here:

<http://web.mit.edu/6.863/spring2009/writeups/lab2/>

Please include 6.863 Lab 2 as the “Subject:” line of your email. As usual, you may collaborate with whomever you wish; just please note the names of your collaborators in your report. Your report should be recognizably your own work.

Initial Preparation: First, be sure you have Python with NLTK and Numpy installed as for Laboratory 1. Then, download the software for this lab from:

<http://web.mit.edu/6.863/spring2009/code/lab2.zip>

Some of the tools you use in this lab will need to be run on Athena.

1 Context-free parsing and Stack-based parsing

Background reading: Read chapter 9 of the NLTK book that we have provided:

<http://web.mit.edu/~6.863/www/spring2009/labs/nltk-ch9.pdf>

Warning: please make sure you read the pdf of the NLTK chapter that we provide in the url specified above. While the material is also covered in the online version of the NLTK book (as chapter 8), the newer online revision do NOT contain the exercises and descriptions required for this Laboratory.

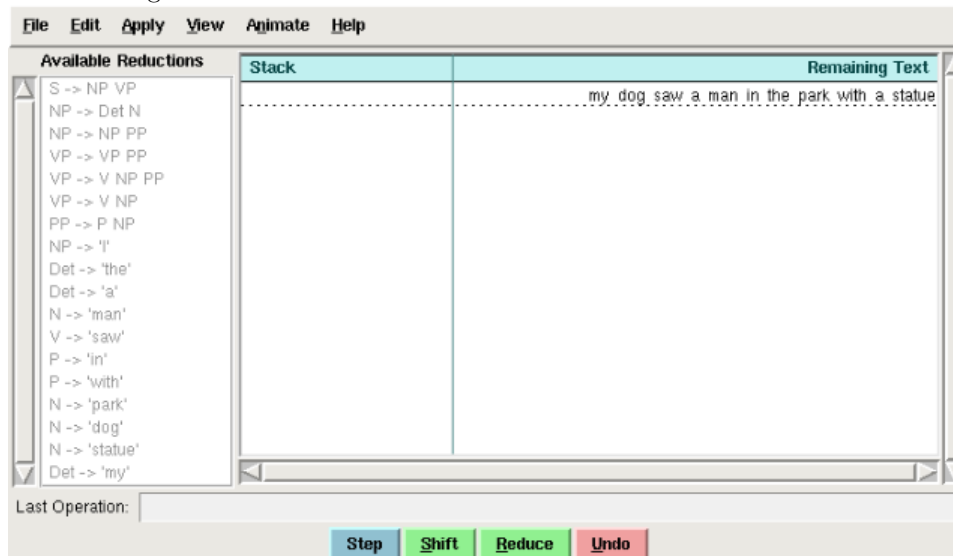
Additional reference material on context-free parsing is available in the Jurafsky and Martin textbook, chapter 13 in its published version, and in draft form here:

<http://www.mit.edu/~6.863/spring2009/jmnew/12.pdf>

The following commands will bring up a GUI with a simple, interactive shift-reduce parser:

```
>>> from nltk.app import srparser_app
>>> srparser_app.app()
```

You should get a window that looks like this:



The pane on the right contains the stack and the remaining input. The pane on the left contains the entire grammar, as a list of production rules.

You can run the parser by clicking the button labeled “Step” repeatedly. Each step will animate one step of the parser, either shifting a word onto the stack or reducing two subtrees on the stack into a new subtree. Which action it has just carried out is displayed at the lower left, in the “Last Operation” panel.

Try stepping through the parser, using the provided example sentence, “my dog saw a man in the park with a statue”. Click on “Step” until the parser gets stuck in a situation where it cannot shift, because the input has no tokens left, and it cannot reduce, because there is no production that involves the items on the stack. As we demonstrated in class, there is not really a way to get any thing parsed if you used a pure “always shift before reduce” policy in terms of this demo software. You get stuck immediately. For each symbol shifted you need to immediately reduce it to get the POS tag, otherwise the parse will guarantee to fail. In effect then, we should state that a “shift” operation really consists of two operations: a shift and an immediate reduce to get the part of speech tag. Even so, it is still easy to run into some state where the parser can no longer proceed, because of some incorrect choice between a **shift** and a **reduce** action. This is called a *shift-reduce conflict*. To get around this parsing misstep, you can now back up the parser by repeatedly hitting the “Undo” button.

Try this for yourself: making sure that you always follow a shift by a reduce in order to get a part of speech tag, see how to force the parser into some corner where neither a shift nor a reduce action will lead to further processing and production of a parse tree that spans the entire sentence. Then, back up to the point at which the incorrect choice was made, and click “Shift” or “Reduce” yourself to make the correct choice.

Problem 1.1. By making manual decisions to shift or reduce when necessary, obtain a parse tree for the whole sentence. Include this parse tree in your lab report.

We therefore see that context-free parsers in general must operate non-deterministically – that is, they must guess. This is a primary source of computational difficulty in natural language processing. Since no physical machine can operate non-deterministically, we must somehow either dodge this problem (use an oracle, order actions probabilistically, figure out how people operate...), or else simulate a nondeterministic machine.

For the moment, we will pursue another plan: developing a way to “always make the right choice” – that is, a parsing strategy that orders choices systematically when more than one action can apply at a given step. In a shift-reduce parser, there are only two possible such conflicts:

- A shift-reduce conflict, in which it is ambiguous whether the parser should shift a new word onto the stack or reduce two items on the stack
- A reduce-reduce conflict, in which there is more than one way to reduce the items on the stack into a new nonterminal

A **strategy** is a systematic resolution of these conflicts imposed on the parser's actions. For example, we could decide to always resolve shift-reduce conflicts in favor of **shift** actions – if a conflict occurs, always shift an item onto the stack, and do not reduce – or we could decide to do the opposite. We denote **shift** > **reduce** as the strategy in which shift takes precedence over reduce, and **reduce** > **shift** as the strategy in which reduce takes precedence over shift.

Problem 1.2. How would you characterize the strategy that the parser is automatically using? Is it **shift** > **reduce** or **reduce** > **shift**?

Problem 1.3. The choice of strategy affects the *shape* of the resulting parse tree. Parse trees can be **bushy** – flatter rather than deeper – or they can be **straggly** – branching deeply to the right or left. Characterize which strategy leads to which type of parse tree in these terms.

Problem 1.4. The example sentence has multiple possible parses. Give a complete parse tree for the sentence that is as bushy as possible, and one that is as straggly as possible. You may, of course, use the shift-reduce parser to help you do this.

Problem 1.5. Putting aside issues about “meaning,” people seem to have definite preferences in the way that they construct parse trees for sentences such as “my dog saw a man in the park with a statue on the grass behind the fence”. Does this preference seem to follow **shift** > **reduce**, **reduce** > **shift**, or neither? Can you justify this in terms of what they put on the stack?

While shift-reduce parsers are hindered by the ambiguity of natural language, they are often used to parse computer languages, where they are called LR parsers. (See http://en.wikipedia.org/wiki/LR_parsing.)

Parsers for computer languages still need to have strategies to resolve conflicts. A very common one is to decide based on the nonterminal on top of the stack and the next token of input that has not yet been shifted. This is known as *simple LR parsing with one token of lookahead*, or LALR(1), and it can be implemented very efficiently with lookup tables on computers. (The “LR” is an acronym invented by Knuth, who first developed this approach: it stands for “parsing a sentence **L**eft to **R**ight, building a **L**eftmost derivation in **R**everse. (The usual cute Knuthian word plays appear.)

An example of using LALR(1) for parsing English might be that when you are parsing a noun phrase, you want to reduce (finishing the noun phrase) if the next word is a verb, but shift if the next word is a preposition (so it can be attached to the noun phrase).

Problem 1.6. Give an example of a conflict in this toy grammar that would not be resolved by the LALR(1) approach.

Interesting generalizations of this strategy are possible. In his thesis work, Marcus (1980) suggested that this one token of lookahead be extended to lookahead for an entire *phrase*, and that the human sentence processor could be so characterized. Marcus's idea was that his method could distinguish between cases like the following, where the parser must decide whether *have* is an auxiliary verb or a main verb:

1. Have the students taken the exam (*Have* is an auxiliary verb, and the sentence is a Yes-No question)
2. Have the students take the exam (*Have* is a main verb, part of an imperative sentence, e.g., “You have the students take the exam.”)

Note that the “disambiguating” material in this case could lie arbitrarily far ahead in terms of number of word *tokens*: it is the tense of the verb (*taken* vs. *take*), but it is only one *phrase* ahead - if the parser could look past the next whole Noun phrase (“the students...”) it could see this disambiguating cue. Marcus claimed this is how human parsing worked. We'll talk a bit about this in a later lecture, or you can read his book, *A Theory of Syntactic Recognition for Natural Language*.

2 Table-driven methods: Chart parsing

Preparation: Please read the provided NTLK book chapter on chart parsing. This material is also covered in section 13.4 of the published JM book (and 12.4 of the on-line draft version).

This section guides you in completing the programming of a simplified chart parser. It is unable to handle phrases that are ambiguous, retaining just one parse for each subsequence of words, and it does not output a parse tree. Your job will be to finish this last part: outputting a parse tree.

We're giving you the basic code from the NLTK documentation, cleaned up slightly, in the `lab2.zip` package. Instead of including the grammar as a literal string, this code loads it from the local file `example1.cfg`, using the versatile `nltk.data.load` function:

```
>>> grammar = nltk.data.load("file:example1.cfg")
```

Problem 2.1. Finish exercise 4 in section 9.2.3 from the above reading preparation, which asks you to augment the code in `simple_chart.py` with the ability to construct a parse tree from a chart. Please include your code changes in your write-up.

The `nltk.Tree` data structure, described in the following url, will be helpful:

<http://web.mit.edu/~6.863/www/spring2009/labs/nltk-tree-pages.pdf>

Here is an example of building up a tree from pieces:

```
>>> tree1 = nltk.Tree('NP', ['John'])
>>> tree2 = nltk.Tree('NP', ['the', 'man'])
>>> tree3 = nltk.Tree('VP', ['saw', tree2])
>>> tree4 = nltk.Tree('S', [tree1, tree3])
>>> print tree4
(S (NP John) (VP saw (NP the man)))
```

To show that your program works, for the given sentence (“the kids opened the box on the floor”), your code should output one of the following trees:

```
(S
  (NP (Det the) (N kids))
  (VP
    (V opened)
    (NP (NP (Det the) (N box)) (PP (P on) (NP (Det the) (N floor))))))

(S
  (NP (Det the) (N kids))
  (VP
    (VP (V opened) (NP (Det the) (N box)))
    (PP (P on) (NP (Det the) (N floor)))))
```

Include a transcript of your program’s output along with your write-up.

Note that NLTK’s chart parser module is more robust than the simple parser you worked with above. You can access it by running code like the following. Here we have set the parser to use the ‘top-down’ strategy for finding possible parses. There are two other strategies that NLTK provides, ‘bottom-up’, and the more sophisticated ‘Earley parser’ strategy.

```

from nltk.parse import chart
sentence = "the kids opened the box on the floor".split()
strategy = nltk.parse.TD_STRATEGY
parser = nltk.ChartParser(grammar, strategy)
parsed_trees = parser.nbest_parse(sentence)
for tree in parsed_trees:
    print tree

```

Problem 2.2.

Preparation: Now read section 9.3 of the provided draft NLTK book chapter, describing the bottom-up, top-down, and Earley strategies for chart parsing. Experiment with the gui interface if you want. Note that you launch the demo gui this way:

```

import nltk
from nltk.app import chartparser_app
chartparser_app.app()

```

The complexity of chart parsing is directly related to the number of edges that can ever get added to the chart (relative to a particular sentence length and a grammar). Any time the parser adds a chart edge (equivalently, table entry) that never participates in a final, correct parse, it has done unnecessary, duplicate work. To operate most efficiently, it should avoid such unnecessary work as much as possible. In this problem we examine this particular issue briefly.

Use the text-mode demo, `nltk.parse.chart.demo()`, to parse a sentence using the top-down, bottom-up, and Earley strategies. Out of all three strategies, what kinds of edges are added *only* by the top-down parser? What kinds of edges are added *only* by the bottom-up parser?

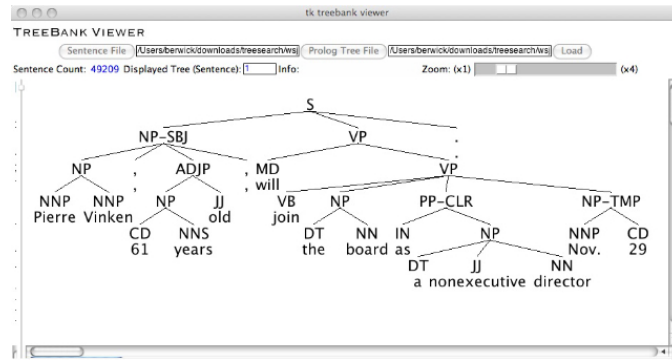
Problem 2.3. In fact there is a whole range of strategies possible when doing chart parsing, some of which are more tailored to languages like English (rather than, say, German or Japanese). Find some example edges produced by the Earley parser that seem unnecessary for completing the demo parse. Describe briefly a modification to the Earley parsing strategy that would prevent these edges from being produced, while still producing all the correct parse trees. (One possible modification is mentioned in the NLTK chapter 9 reading).

3 Scaling up: Using Larger Grammars

Of course, this is all very well and good for “toy examples”, but what about the real world? In this part of the Laboratory, we will begin our exploration of the challenges faced when dealing with a fuller range of English, and so much larger grammars. As a resource we will use what has become something of a standard for testing (and machine learning) for parsing, the 49,208 sentences drawn from the Wall Street Journal in the late 1980s. These sentences were then (mostly) hand-parsed (“bracketed”) and given annotated parse trees by a group at the University of Pennsylvania. This corpus is variously named “the Penn Treebank,” (PTB) sometimes the “WSJ corpus” (after *Wall Street Journal*); or, as NLTK calls it, simply “treebank.” For example, here is just one sentence from this corpus:

Pierre Vinken , 61 years old , will join the board as a nonexecutive director Nov. 29 .

And here is the corresponding hand-annotated parse tree for this sentence. The parse tree makes use of a standard set of phrase names (nonterminals) and categories for words (part of speech tags, abbreviated POS). Note that the POS tags are those symbols that immediately dominate the words. We will have more to say about tags and nonterminal names in later lectures and Laboratories, but after the picture we give a brief table of the phrase names and part of speech tags used in this example.



Phrase (nonterminal)	
S	Sentence or clause
NP-SBJ	Noun Phrase used as a Subject
NP	Noun Phrase
ADJP	Adjective Phrase
VP	Verb Phrase
PP-CLR	Prepositional phrase "closely related" to the higher phrase
NP-TMP	Noun Phrase indicating when, how often, how long
Part of speech tag	
NNP	Proper noun, singular
CD	Cardinal determiner (number)
NNS	Proper noun, plural
MD	Modal auxiliary verb
VB	Verb, root
DT	Determiner
NN	Noun, singular or mass
IN	Preposition
JJ	Adjective

And here is how you can access sentences in a 10% sample of this corpus from nltk:

```
>>> from nltk.corpus import treebank
>>> t = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> print t
(S
 (NP-SBJ
  (NP (NNP Pierre) (NNP Vinken))
  (, ,)
  (ADJP (NP (CD 61) (NNS years)) (JJ old))
  (, ,))
 (VP
  (MD will)
  (VP
   (VB join)
   (NP (DT the) (NN board))
   (PP-CLR
    (IN as)
    (NP (DT a) (JJ nonexecutive) (NN director)))
   (NP-TMP (NNP Nov.) (CD 29))))
 (. .))
```

3.1 Running time, ambiguity, and grammar size

In class we show that the worst-case running time of chart parsers is $O(|G|^i n^3)$, where $|G|$ is the size of the grammar (simply a sum of all the symbols it takes to write down the grammar, see below), i is at most 2, and n is the length of the input sentence in tokens (or words).¹ Thus the running time is at worst quadratic in the grammar size and cubic in the sentence length. This means that the size of the grammar, though it is in fact a mere “constant”, plays a crucial role in the running time of a parsing system for natural language as soon as we move to any kind of grammar that is more than a toy system. In fact, in practical parsing, constants like this are everywhere. The aim of the following problems are to give you some appreciation of such factors, to see why in practice one is almost forced to some method of probabilistic pruning, as described in the next section.

Let us now look more carefully at calculating grammar size. The size of the toy grammar “demo” used in section 1 of the Lab for shift-reduce parsing can be calculated as follows, where the right-hand arrow \rightarrow counts as one symbol, the vertical bar $|$ counts as 1 symbol and means “or”, and every other name made up of 1 or more characters, like NP or ‘the’ counts as 1 symbol. Thus the first line in the grammar below uses 4 symbols, while the second uses 7. Continuing, the right hand number given below gives the number of symbols for that line:

S \rightarrow NP VP	4
NP \rightarrow Det N NP PP	7
VP \rightarrow VP PP V NP PP V NP	11
PP \rightarrow P NP	4
Det \rightarrow ‘the’ ‘a’	5
N \rightarrow ‘man’	3
V \rightarrow ‘saw’	3
P \rightarrow ‘in’ ‘with’	5
N \rightarrow ‘park’ ‘dog’ ‘statue’	7
Det \rightarrow ‘my’	3

If you add up the column of numbers, you get the total grammar size, which is 52.

Now let’s experiment with a much larger grammar. We will use a grammar that has been trained on 50% of the Penn Treebank (sentences containing coordination words like “and” were excluded. To understand why, refer to the “Martha Stewart’s Revenge” example that was given in lecture 5, repeated in lecture 6.). This file is available as the file `wsj.cfg` in the `lab2.zip` package that you downloaded earlier. Here the “training” is very simple: we simply examined the parse trees to see which rules were ever used to build any part of any parse tree and added those to the grammar; you’ll do something a bit smarter in the next section of the Lab. To get some idea of this grammar’s size, you can examine it using an editor like emacs and see that it contains approximately 10,000 individual rules. So this is roughly 2 orders of magnitude larger than the toy grammar you used earlier. But more precisely...

Problem 3.1. What is the size of the `wsj.cfg` grammar, using the calculation method we gave above? (Yes; this is a straightforward question, for warm-up. You can use any method you want to do this calculation. Hint: There is a simple 2-letter unix command that can tell you the number of symbols in a grammar file.)

Now let’s investigate how this larger grammar affects parsing. First, let’s load the larger grammar and see how parsing time is affected for just a simple example sentence, *John is happy*.

To load the grammar and set up a simple Earley parser (without any tracing turned on) we simply do the following, making sure the `wsj.cfg` is in your current directory path. Note that even for this short sentence you will have to wait a few minutes for the parse to finish – that is why we do not turn on any tracing, though you might want to set `trace=1` in the definition of the parser to see why we say “a few minutes.”

We have provided you a scripts to run parses.

¹Here we use the standard “Big Oh” notation to say that the growth of some function is bounded “from above”; i.e., for real-valued functions f and g , $f(x) = O(g(x))$ $x \rightarrow \infty$ iff there exists a positive real number M and a real number x_0 such that for $x > x_0$, $|f(x)| \leq M|g(x)|$. For additional details, see http://en.wikipedia.org/wiki/Big_O_notation.

```
cfg_parse.py <cfg file> <sentence file> <sentence number> [verbose]
```

For example, running:

```
cfg_parse.py wsj.cfg sentences.txt 0
```

will parse the first (0th) sentence in `sentences.txt` using the `wsj.cfg` grammar file. and output the number of parses, and time taken to generate the parse.

Running this script with the verbose option will also print out the parse trees. Before you print them out, you might want to try running the script without the verbose option to see how many there are. You might be surprised by this number.

Problem 3.2. How many parse trees are there for this first, simple sentence?

Problem 3.3. Are there *any* of the first 100 trees that correspond to what the “correct” parse for “John is happy” ought to be, which something like the following? (You might want to write a program to help with the search or modify `cfg_parse.py`.) If you can find this one, how far down the first 100 trees is it?

```
(S
  (NP (NPR (NNP John)))
  (VP (VBZ is) (ADJPPRD (JJ happy)))
  (PUNCpoint .))
```

Problem 3.4. You’ll notice that `sentences.txt` contain 2 other test sentences namely: `John saw the companies with the stock .` and `The very biggest companies are not likely to go under .` If you attempt to parse these with the `wsj.cfg` grammar using `cfg_parse.py` you’ll find that they will not finish, unless you are using a very very powerful machine with lots of RAM.

To parse these test sentences, we have created a series of stripped-down “partial” grammars from the `wsj.cfg` grammar. These grammars were generated by taking the minimum set of productions to parse 3 test sentences and then adding to that a random fraction of productions from the `wsj` grammar. These partial grammars are found in the `partial_wsj` subfolder in the provided zip package. The three digit number in the grammar file name indicates the percentage of the `wsj` grammar that was included in the stripped-down grammar.

To test the relationship between grammar size and parsing time, we have provided another script that can parse a directory of grammar files:

```
cfg_parse_runner.py <directory with .cfg files> <sentence file> <sentence number>
```

For example, running:

```
cfg_parse_runner.py partial_wsj sentences.txt 0
```

will parse the first (0th) sentence using successive grammars on the `partial_wsj` directory. Please plot the output of the script, that is, grammar size versus number of parses, and grammar size versus running time. What do you observe – how does the grammar size vary with the plotted relations? Please include your plots with your answers.

Now run the script for sentences 1 and 2 from `sentences.txt`. Note that you will almost certainly have to terminate your program manually when parsing takes too long. Plot what data points you were able to obtain before you terminated the program.

4 Probabilistic context-free parsing

4.1 Using probabilistic context-free grammars

Preparation: Read section 9.2-9.3 of the NLTK book chapter, on how to define and use probabilistic context-free grammars using NLTK. For additional reference, see the published JM text, chapter 14 up through section 14.6.

Before trying you hand at a problem we would like you to appreciate that by assigning probabilities to context-free rules, one can construct a chart parser that works much faster than the “all possible parses” method in the previous section, because one can prune away low probability rule combinations. As an example, the following code fragment is extracted from `pcfg-parse-wsj.py` in the package we have provided for this laboratory. It uses an A* parsing method, similar to the A* path finding methods you learned about in 6.034, and so finds a single, most likely (“optimal”) parse of a sentence with respect to a given probabilistic context-free grammar. Since it does not have to keep all the parses around, it works much faster. As you may recall from 6.034, for this A* method to work, one needs two things: (1) a measure of ‘distance so far’, which in this case is just ‘probability of the parse so far’, where we measure this just by multiplying together the individual rule probabilities that were used to build the parse so far; and (2) a (valid) estimate of the ‘distance remaining’, where a valid estimate must be an *underestimate* of the true distance – once again, we must map the notion of ‘distance’ somehow onto ‘probability’, a matter which we will not take up here. (There is one other condition to impose in order to have a *valid* estimate of the distance remaining – can you recall from 6.034 what this other condition is? We’ll describe A* parsing in more detail in later lectures.)

In any event, this A* search method has been implemented in NLTK as a so-called `ViterbiParser`, this being simply another name for A*. You can invoke it this way.

```
>>> def pcfg_chartparser(grammarfile):
    f = open(grammarfile)
    grammar = f.read()
    f.close()
    return nltk.ViterbiParser(nltk.parse_pcfg(grammar))
>>> pcfg_chartparser("wsjp.cfg")
<ViterbiParser for <Grammar with 10669 productions>>
>>> grammarp = pcfg_chartparser("wsjp.cfg")
print grammarp.parse(sent1)
(S
  (NP (NPR (NNP John)))
  (VP (VBZ is) (ADJPPRD (JJ happy)))
  (PUNCpoint .)) (p=1.23978875209e-10)
```

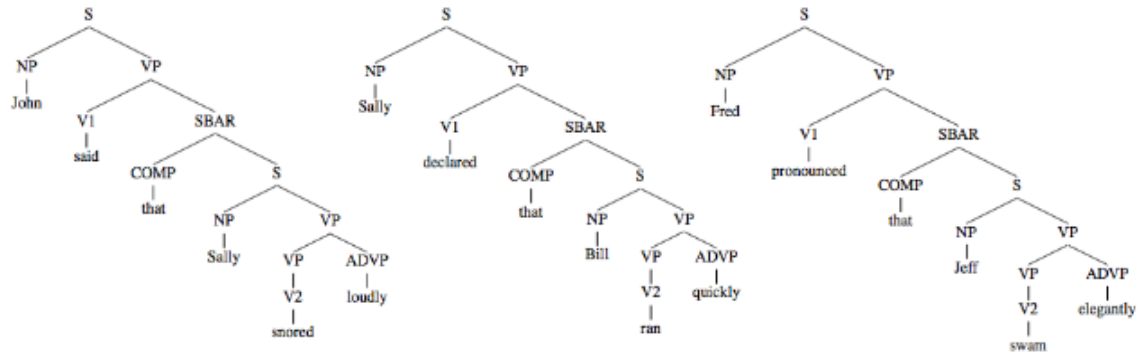
Problem 4.1. Again parse the sentences from `sentences.txt`, but this time using the A* parsing method and the probabilistic grammar `wsjp.cfg`. Compare your parsing times now against those from using just standard chart parsing in the previous section. You should have running times for sentences 0, 1, and 2 using the chart parser with `partial-wsj/g1.010.cfg` from the last exercise. Plot sentence length versus running time, for all 3 test sentences using the chart parser and then using the probabilistic Viterbi parser. In a sentence or two describe your results, and comment as to whether the plotted relationship is what you expected, given the algorithmic analysis.

4.2 Learning probabilistic context-free grammars

Preparation: Read section 9.5 of chapter 9 from the draft NLTK book obtained as above, on how to train probabilistic context-free grammars (PCFGs) from example parse trees.

We’ve written some utility functions for learning PCFGs; they’re included in the package as `learn_pcfg.py`. You should read this file and its comments; it will make your life easier. This code can learn from a list of trees you provide, or from sections of the Penn Treebank.

Problem 4.2. Alyssa Hacker decides to build a treebank. She finally produces a corpus which contains the following three parse trees, all of which have the same structure:



Clearly, this is an important data resource and it deserves to be represented in NLTK. So we've defined these trees in the variable `three_trees` in `learn_pcfg.py`.

You decide to train a PCFG based on Alyssa's treebank, so that you can put in similar sentences and get similar parse trees. Use NLTK and our utility functions to train the maximum-likelihood PCFG for this treebank, and `print` it to see its rules. What are the rules for expanding the nonterminal `VP` and their associated probabilities?

Problem 4.3. Using this PCFG, find the **two** most likely parses for the sentence "Jeff pronounced that Fred snored loudly". What are their probabilities?

Problem 4.4. You are surprised to find that this grammar produces ambiguous results, even though Alyssa's corpus was so consistent. One of the parse trees has a structure never seen in the three examples: the adverb *loudly* attaches to the "higher" verb, *pronounced*, instead of to *snored*.

You don't want this ambiguity; you want trees with high attachment to have zero probability, because they never appear in the corpus.

Make a small modification to the corpus, in a way that may slightly increase the number of non-terminals in the grammar, so that the resulting grammar will only give you parses with low attachment. What change did you make? What is the probability of the correct parse tree now?

4.3 Learning from a Large Corpus

Now it's time to move on to a bigger corpus. If you have imported the Treebank corpus with `from nltk.corpus import treebank`, the function `treebank.parsed_sents()` will give you parse trees representing a 10% sample of the Penn Treebank's *Wall Street Journal* training data.

Problem 4.5. Train a probabilistic grammar from this 10% sample. How many rules does the grammar have? (Using `print` on a grammar will display a count of the total number of rules, followed by all of the rules. You can also examine `len(grammar.productions())`.)

That's rather a lot of rules, and many of them are highly artificial rules that result from the Chomsky normal form translation (binary-branching form, aside from terminal, part of speech tags). If you used these in a parser, as we saw in the previous section, it would be bogged down in rules that were almost never used. A trick called *Markov smoothing*, however, can combine some of these artificial rules together. You can read about it in the API documentation at:

<http://nltk.googlecode.com/svn/trunk/doc/api/nltk.treetransforms-module.html>

Problem 4.6. Retrain your grammar using Markov order-1 smoothing. How many rules does the grammar have now?

For testing purposes, we will use four sentences that occur outside of this 10% sample. Figure 1, on page 11, shows the sentences in their gold-standard parse trees.

Figure 1: The parse trees for our four test sentences.

1. (S
 (NP-SBJ (DT The) (NN luxury) (NN auto) (NN maker))
 (NP-TMP (JJ last) (NN year))
 (VP (VBD sold)
 (NP (CD 1,000) (NNS cars))
 (PP-LOC (IN in)
 (NP (DT the) (NNP U.S.)))))

2. (S
 (NP-SBJ (NNP Bell) (NNP Industries) (NNP Inc.))
 (VP (VBD increased)
 (NP (PRP\$ its) (NN quarterly))
 (PP-DIR (TO to)
 (NP (CD 10) (NNS cents)))
 (PP-DIR (IN from)
 (NP
 (NP (CD seven) (NNS cents))
 (NP-ADV (DT a) (NN share)))))
 (. .))

3. (S
 (NP-SBJ (DT The) (NN dispute))
 (VP (VBZ shows)
 (ADVP-MNR (RB clearly))
 (NP
 (NP (DT the) (JJ global) (NN power))
 (PP (IN of)
 (NP
 (NP (NNP Japan) (POS 's))
 (JJ financial) (NNS titans)))))
 (. .))

4. (SQ (VBD Was)
 (NP-SBJ (DT this))
 (SBAR-PRD
 (WHADVP-1 (WRB why))
 (S
 (NP-SBJ
 (NP (DT some))
 (PP (IN of)
 (NP (DT the) (NN audience)))))
 (VP (VBD departed)
 (PP-TMP (IN before) (CC or) (IN during)
 (NP (DT the) (JJ second) (NN half)))))
 (. ?))

To use these sentences as input to a statistical parser, they need to be *tokenized*: there need to be spaces between everything that the parser considers as a separate token, not just where spaces would be used in natural English. For example, the tokenized version of sentence 3 is:

The dispute shows clearly the global power of Japan 's financial titans .

These four sentences appear, appropriately tokenized, as the variable `sentences` in `learn_pcfg.py`, so you can refer to them as `sentences[0]` through `sentences[3]`.

Problem 4.7. Use your smoothed grammar to parse these sentences. About what percentage of constituents does this parser correctly identify in the first three sentences? One way to measure this is to count the number of constituents in the correct (reference) tree, and the parsed tree then identify how many constituents they have in common. A common constituent would have the same nonterminal spanning the same words. We have provided a small program `tree_eval.py` to help you compute this constituent set intersection:

```
python tree_eval.py <reference.txt> <parsed.txt>
```

Here, `reference.txt` is a text file containing the s-expression of the correct tree, sometimes called the “gold standard” tree, while `parsed.txt` is a text file containing the s-expression of the tree output by the parser. You may use functions in `tree_eval` directly to write a program that computes the precision and recall of the parsed trees relative to the correct trees.

Problem 4.8. Why can’t the current grammar parse the fourth sentence? You should observe at least two problems. Suggest ways to fix each of these problems and make the parser less fragile. (Be brief; You do not need to implement your fixes.)

4.4 Optional: Accessing the entire corpus

If you want to experiment with the entire Wall Street Journal training set, instead of a 10% sample of it, we have it installed on Athena. You can access it with:

```
trees = treebank.parsed_sents('wsj-02-21.mrg')
```

You can install it on your own machine as well. Find the directory where your NLTK data is installed, such as `/usr/share/nltk/data`. That directory should have a subdirectory called `corpora/treebank`. Copy `/mit/6.863/tools/wsj-02-21.mrg` from Athena into that directory, and NLTK will then be able to find it.

However, you may find that using all this data in a simple NLTK parser makes it take a considerable amount of time to run.