Homework 4: CKY Algorithm and Dependency Parsing

1: CKY Algorithm (30 points)

In this section, you will implement the CKY algorithm for an unweighted CFG. See the starter code cky.py.

Question 1.1 (8 points)

Implement the acceptance version of CKY as `cky_acceptance()`, which returns True if there is a `S` covering the entire sentence. Does it return True or False for the following sentences? Please `pprint()` the chart cells for each case as well.

- the the
- the table attacked a dog
- the cat

Hint: A simple way to implement the chart cells is by maintaining a list of nonterminals at the span. This list represents all possible nonterminals over that span.

Hint: `pprint()`ing the CKY chart cells may be useful for debugging.

Hint: Python dictionaries allow tuples as keys. For example, `d={}; d[(3,4)] = []`. A slight shortcut is that `d[3,4]` means the same thing as `d[(3,4)]`.

In [8]:

```
import cky
from pprint import pprint
print "Grammar rules in tuple form:"
pprint(cky.grammar_rules)
print "Rule parents indexed by children:"
pprint(cky.possible_parents_for_children)
```

```
Grammar rules in tuple form:
[('S', ('NPZ', 'VP')),
 ('S', ('NP', 'VBZ')),
 ('NP', ('Det', 'Noun')),
 ('PP', ('Prep', 'NP')),
 ('NP', ('NP', 'PP')),
 ('VP', ('VP', 'PP')),
 ('NPZ', ('Det', 'Nouns')),
 ('VP', ('Verb', 'NP')),
 ('VBZ', ('Verbs', 'NP'))]
Rule parents indexed by children:
{('Det', 'Noun'): ['NP'],
 ('Det', 'Nouns'): ['NPZ'],
 ('NP', 'PP'): ['NP'],
 ('NP', 'VBZ'): ['S'],
 ('NPZ', 'VP'): ['S'],
```

```
 ('Prep', 'NP'): ['PP'],
 ('VP', 'PP'): ['VP'],
 ('Verb', 'NP'): ['VP'],
 ('Verbs', 'NP'): ['VBZ']}
```

In [9]:

```
#Print the result here
import cky;reload(cky)
print "the the"
print cky.cky_acceptance(["the", "the"])
print "the table attacked a dog"
print cky.cky_acceptance(["the", "table", "attacked", "a", "dog"])
print "the cat"
print cky.cky_acceptance(["the", "cat"])
```

```
the the
{(0, 1): ['Det'], (1, 2): ['Det'], (0, 2): []}
False
the table attacked a dog
((0, 2, 'NP'), (2, 5, 'VBZ'))
{(0, 1): ['Det'], (1, 2): ['Noun'], (2, 5): ['VBZ'], (1, 3): [], (4, 5): [
'Noun'], (1, 4): [], (2, 4): [], (1, 5): [], (0, 5): ['S'], (0, 4): [], (2
, 3): ['Verbs'], (0, 3): [], (3, 4): ['Det'], (0, 2): ['NP'], (3, 5):
['NP']}
True
the cat
{(0, 1): ['Det'], (1, 2): ['Noun'], (0, 2): ['NP']}
False
```

Question 1.2 (15 points)

Implement the parsing version of CKY, which returns one of the legal parses for the sentence (and returns None if there are none). If there are multiple real parses, we don't care which one you print. Implement this as `cky_parse()`. You probably want to start by copying your `cky_acceptance()` answer and modifying it. Have it return the parse in the following format, using nested lists to represent the tree (this is a simple Python variant of the Lisp-style S-expression that's usually used for this.)

```
['S',
        [['NP', [['Det', 'the'], ['Noun', 'cat']]],
         ['VP', [['Verb', 'attacked'],
                 ['NP', [['Det', 'the'], ['Noun', 'food']]]]]]]
```

Print out the parses for the following sentences.

- the cat saw a dog
- the cat saw a dog in a table
- the cat with a table attacked the food

Hint: In the chart cells, you will now have to store backpointers as well. One way to do it is to store a list of tuples, each of which is `(nonterminal, splitpoint, leftchild nonterm, rightchild nonterm)`. For example, if the state `('NP', 3, 'Det', 'Noun')` is in the cell for span (2,4), that means this is a chart state of symbol NP, which came from a `Det` at position (2,3)

span (2, 4), that means this is a chart state of symbol NP , which came from a Det at position (2,3) and a Noun at position (3,4).

Hint: It may be useful to use a recursive function for the backtrace.

In [10]:

```
# Output the results for each sentence.
#TODO: Print out the parse tree for each of the three sentence
import cky;reload(cky)
#pprint(
cky.cky_parse(['the','cat','with','a','table','attacked','the','food']) )
pprint( cky.cky_parse(['the','cat','attacked','the','food']) )
print "*" * 50
```

```
['S', [['NP', [['Det', 'the'], ['Noun', 'cat']]], ['VBZ', [['Verbs', 'attac
ked'], ['NP', [['Det', 'the'], ['Noun', 'food']]]]]]]
True
**************************************************
```

Question 1.3 (7 points)

Revise the grammar as follows.

- Add four words to the lexicon: two verbs “attack” and “attacks”, and the nouns “cats” and “dogs”.
- Revise the rules to enforce subject-verb agreement on number.

The new grammar should accept and reject the following sentences. Please run your parser on these sentences and report the parse trees for the accepted ones. Also, describe how you changed the grammar, and why.

ACCEPT: `the cat attacks the dog`
REJECT: `the cat attack the dog`
ACCEPT: `the cats attack the dog`
REJECT: `the cat with the food on a dog attack the dog`

Hint: you will need to introduce new nonterminal symbols, and modify the currently existing ones.

In [11]:

```
# Output the results for each sentence.
#TODO: Print out the parse tree for each of the four sentence
import cky;reload(cky)
pprint( cky.cky_parse(['the','cat','attacks','the','dog']) )
pprint( cky.cky_parse(['the','cat','attack','the','dog']) )
pprint( cky.cky_parse(['the','cats','attack','the','dog']) )
pprint( cky.cky_parse(['the','cat','with', 'the', 'food', 'on','a', 'dog',
'attack','the','dog']) )
print "*" * 50
```

```
['S', [['NP', [['Det', 'the'], ['Noun', 'cat']]], ['VBZ', [['Verbs', 'attac
ks'], ['NP', [['Det', 'the'], ['Noun', 'dog']]]]]]]
True
False
['S', [['NPZ', [['Det', 'the'], ['Nouns', 'cats']]], ['VP', [['Verb', 'atta
ck'], ['NP', [['Det', 'the'], ['Noun', 'dog']]]]]]]
True
```

```
False
**************************************************
```

lexicon = {

```
    'Nouns': set(['cats', 'dogs']),

    'Verbs': set(['attacks', 'attacked']),

    'Noun': set(['cat', 'dog', 'table', 'food']),

    'Verb': set(['saw', 'loved', 'hated', 'attack']),

    'Prep': set(['in', 'of', 'on', 'with']),

    'Det': set(['the', 'a']),
```

}"""

grammar_text = """

S -> NPZ VP

S -> NP VBZ

PP -> Prep NP

NP -> NP PP

VP -> VP PP

NP -> Det Noun

NPZ -> Det Nouns

VP -> Verb NP

VBZ -> Verbs NP

"""

2: Weighted CKY Algorithm (40 points)

In this section you will implement the weighted CKY Algorithm for a Probabilistic CFG. You will have to make modifications to the existing algorithm to account for the probabilities and your parse function should output the most probable parse tree. Please write all your code in `weighted_cky.py` file for this section.

Question 2.1 (7 points)

The CKY Algorithm requires the CFG to be in Chomsky Normal Form. Convert the following CFG into Chomsky Normal Form. (For the sake of uniformity, replace the leftmost pairs of non-terminals with new non-terminal)

S -> Aux NP VP

S -> Aux NP VP
S -> VP
VP -> Verb NP
VP -> VP PP
Verb -> book
Aux -> does

Modified Form

S -> ANP VP
ANP -> Aux NP
S -> Verb NP
S -> VP PP
VP -> Verb NP
VP -> VP PP
Verb -> book
Aux -> does

Question 2.2 (8 points)

We will now implement a weighted CYK algorithm to parse a sentence and return the most probable parse tree. The grammar is defined in `pcfg_grammar_original.txt`. As you can notice, some of the rules are not in CNF. Modify the `pcfg_grammar_modified.txt` file such that all the rules are in Chomsky Normal Form. (For the sake of uniformity, replace the leftmost pairs of non-terminals with new non-terminal)

Note: When transforming the grammar to CNF, must set production probabilities to preserve the probability of derivations.

Question 2.3 (5 points)

Explain briefly what are the changes you made to convert the grammar into CNF Form. Why did you make these changes?

Rules with more than one terminals on the right side have been modified and also unary rules removed.

Question 2.4 (8 points)

Complete the `populate_grammar_rules()` function in the `weighted_cky.py` script. This function will have to read in the grammar rules from `pcfg_grammar_modified.txt` file and populate the `grammar_rules` and `lexicon` data structure. Additionally you would need to store the probability mapping in a suitable data structure.

Hint: You can modify the starter code provided in cky.py for this task.

In [12]:

```
import weighted_cky;reload(weighted_cky)
from weighted_cky import populate_grammar_rules
```

```
populate_grammar_rules()
```

Question 2.5 (12 points)

Implement the weighted parsing version of CKY, which returns the most probable legal parse for the sentence (and returns None if there are none). If there are multiple real parses, this function will always return the most probable parse i.e the one with maximum probability. Complete the `pcky_parse()`. Print the parse tree and the probabilities for the following sentences:

- book the flight through Houston
- include this book
- the the

Hint: You can use the code in `cky_parse()` and modify it to accomodate probabilities and compute the most probable parse.
Note: The topmost cell should contain rules associated with the `S` non terminal, if any.

In [13]:

```
import weighted_cky;reload(weighted_cky)
from weighted_cky import pcky_parse
# Output the results for each sentence.
#TODO: Print out the parse tree for each of the three sentence
pprint( pcky_parse(['the','the']) )
pprint( pcky_parse(['book','the','flight','through','Houston']) )
pprint(pcky_parse(['include','this','book']))
print "*" * 50
```

```
None
'Sentence: the the'
'Not a valid parse tree for this sentence'
False
None
'Sentence: book the flight through Houston'
'Not a valid parse tree for this sentence'
False
None
'Sentence: include this book'
'Not a valid parse tree for this sentence'
False
**************************************************
```

3: Dependency parser output (30 points)

You will conduct manual error analysis of CoreNLP's dependency parser.

Create an English sentence where the parser makes an error, and you know what the correct analysis ought to be, according to the Universal Dependencies grammatical standard. You will want to play around with different sentences, look at their output, and check against the Universal Dependencies annotation standard. The current version of CoreNLP outputs according to the "UD version 1" standard, so please use this page:

- UD v1 homepage
- and in particular, the UD v1 dependency relations list

For quickly looking at things, their online demo may be useful.

However, for this assignment, you need to run the parser to output in "conllu" format, which is human readable. You need to download and run the parser for this. (It requires Java.) Use version 3.8.0 (it should be the current version). You can it working in interactive mode so you can just type sentences into it on the terminal like this:

```
./corenlp.sh -annotators tokenize,ssplit,pos,lemma,depparse -outputF
ormat conllu
[...]
Entering interactive shell. Type q RETURN or EOF to quit.
NLP>
```

For example then you can type

```
NLP> I saw a cat.
1       I       I       _       PRP     _       2       nsubj   _
_
2       saw     see     _       VBD     _       0       root    _
_
3       a       a       _       DT      _       4       det     _
_
4       cat     cat     _       NN      _       2       dobj    _
_
5       .       .       _       .       _       2       punct   _
_
```

You can also use the `-inputFile` flag if you'd rather give it a whole file at once.

As you can see in the parser documentation, the 7th and 8th columns describe the dependency edge for the word's parent (a.k.a governor): it has the index of its parent, and the edge label (a.k.a. the relation). For example, this parse contains the dependency edge *nsubj(saw:2, I:1)* meaning that "I" is the subject of "saw".

Question 3.1: Once you've decided your sentence, please put the conllu-formatted parser output below in the markdown triple-quoted area. Please be very careful where it goes since we will use a script to pull it out.

PARSE GOES BELOW HERE

```
1   I   I   _   PRP _   2   nsubj   _   _
2   live    live    _   VBP _   0   root    _
3   in  in  _   IN  _   2   case    _   _
4   a   a   _   DT  _   2   det _   _
5   4   4   _   CD  _   2   nummod  _   _
6   bedroom bedroom _ NN _  0   compound
7   house   house   _   NN  _   4   nmod:in _   _
```

PARSE GOES ABOVE HERE

Question 3.2: Please describe the error you found. Also give a citation and link to the relevant part of the UD documentation describing one of the relations that the parser predicted in error, or did something wrong with.

The term 4 bedroom is a single combined adjective for house. In this case 4 is a nummod and bedroom is a compiund since the parser is confused about the amod identifier.

http://universaldependencies.org/docsv1/u/dep/amod.html
http://universaldependencies.org/docsv1/u/dep/nmod.html

Question 3.3: Please give correct that error in the parse . Put your corrected parse, again in that conllu textual format, below. You should take a copy of the output and manually change some of the 7th/8th dependency edge columns.

PARSE GOES BELOW HERE

```
1   I   I   _   PRP _   2   nsubj   _   _
2   live    live    _   VBP _   0   root    _
3   in  in  _   IN  _   2   case    _   _
4   a   a   _   DT  _   2   det _   _
5   4-bedroom   4-bedroom   _   NN  _   2   compound    _   _
6   house   house   _   NN  _   4   nmod:in _   _
```

PARSE GOES ABOVE HERE

Question 3.4: Please describe your correction and why it solves the error.

Here by making 4-bedroom, a hyphenated word it is considered to be a compound. In the previous case, bedroom was a compound which was wrong. Now 4-bedroom is a amod adjective compound for the word "house".