50.040 Natural Language Processing (Summer 2020) Homework 2

Due

STUDENT ID: 1003014

Name: Antonio Miguel Canlas Quizon

Students with whom you have discussed (if any):

In [1]:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6
qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%
b&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2
www.googleapis.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly
ttps%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/gdrive
```

In [2]:

```
% pwd
% ls
% cd /content/gdrive/'My Drive'/'NLP'/'HW2'
% ls
```

```
gdrive/  sample_data/
/content/gdrive/My Drive/NLP/HW2
homework2.ipynb  hw2.pdf  imgs/  __MACOSX/
```

In [3]:

```
import copy
from collections import Counter
from nltk.tree import Tree
from nltk import Nonterminal
from nltk.corpus import LazyCorpusLoader, BracketParseCorpusReader
from collections import defaultdict
import time
```

In [3]:

```

```

In [4]:

```
st = time.time()
```

In [5]:

```
import nltk
nltk.download('treebank')
```

```
[nltk_data] Downloading package treebank to /root/nltk_data...
[nltk_data]   Unzipping corpora/treebank.zip.
```

Out[5]:

```
True
```

In [6]:

```
def set_leave_lower(tree_string):
    if isinstance(tree_string, Tree):
        tree = tree_string
    else:
        tree = Tree.fromstring(tree_string)
    for idx, _ in enumerate(tree.leaves()):
        tree_location = tree.leaf_treeposition(idx)
        non_terminal = tree[tree_location[:-1]]
        non_terminal[0] = non_terminal[0].lower()
    return tree

def get_train_test_data():
    '''
    Load training and test set from nltk corpora
    '''
    train_num = 3900
    test_index = range(10)
    treebank = LazyCorpusLoader('treebank/combined', BracketParseCorpusReader, r'wsj_.*\.mrg')
    cnf_train = treebank.parsed_sents()[:train_num]
    cnf_test = [treebank.parsed_sents()[i+train_num] for i in test_index]
    #Convert to Chomsky norm form, remove auxiliary labels
    cnf_train = [convert2cnf(t) for t in cnf_train]
    cnf_test = [convert2cnf(t) for t in cnf_test]
    return cnf_train, cnf_test
def convert2cnf(original_tree):
    '''
    Chomsky norm form
    '''
    tree = copy.deepcopy(original_tree)

    #Remove cases like NP->DT, VP->NP
    tree.collapse_unary(collapsePOS=True, collapseRoot=True)
    #Convert to Chomsky
    tree.chomsky_normal_form()

    tree = set_leave_lower(tree)
    return tree
```

In [7]:

```
### GET TRAIN/TEST DATA
cnf_train, cnf_test = get_train_test_data()
```

In [8]:

```
cnf_train[0].pprint()
```

```
(S
  (NP-SBJ
    (NP (NNP pierre) (NNP vinken))
    (NP-SBJ|<,-ADJP-,>
      (, ,)
      (NP-SBJ|<ADJP-,>
        (ADJP (NP (CD 61) (NNS years)) (JJ old))
        (, ,))))
  (S|<VP-.>
    (VP
      (MD will)
      (VP
        (VB join)
        (VP|<NP-PP-CLR-NP-TMP>
          (NP (DT the) (NN board))
          (VP|<PP-CLR-NP-TMP>
            (PP-CLR
              (IN as)
              (NP
```

```
                (DT a)
                (NP|<JJ-NN> (JJ nonexecutive) (NN director))))
            (NP-TMP (NNP nov.) (CD 29))))))
    (. .)))
```

Question 1

To better understand PCFG, let’s consider the first parse tree in the training data “cnf_train” as an example. Run the code we have provided for you and then writedown the roles of.productions(), .rhs(), .lhs(), .leaves()in the ipynb notebook.

In [9]:

```python
rules = cnf_train[0].productions()
print(rules, type(rules[0]))
```

```
[S -> NP-SBJ S|<VP-.>, NP-SBJ -> NP NP-SBJ|<,-ADJP-,>, NP -> NNP NNP, NNP -> 'pierre', NNP -> 'vin
ken', NP-SBJ|<,-ADJP-,> -> , NP-SBJ|<ADJP-,>, , -> ',', NP-SBJ|<ADJP-,> -> ADJP ,, ADJP -> NP JJ,
NP -> CD NNS, CD -> '61', NNS -> 'years', JJ -> 'old', , -> ',', S|<VP-.> -> VP ., VP -> MD VP, MD
-> 'will', VP -> VB VP|<NP-PP-CLR-NP-TMP>, VB -> 'join', VP|<NP-PP-CLR-NP-TMP> -> NP VP|<PP-CLR-NP
-TMP>, NP -> DT NN, DT -> 'the', NN -> 'board', VP|<PP-CLR-NP-TMP> -> PP-CLR NP-TMP, PP-CLR -> IN
NP, IN -> 'as', NP -> DT NP|<JJ-NN>, DT -> 'a', NP|<JJ-NN> -> JJ NN, JJ -> 'nonexecutive', NN -> '
director', NP-TMP -> NNP CD, NNP -> 'nov.', CD -> '29', . -> '.'] <class
'nltk.grammar.Production'>
```

In [10]:

```python
rules[4].rhs(), type(rules[0].rhs()[0])
```

Out[10]:

```
(('vinken',), nltk.grammar.Nonterminal)
```

In [11]:

```python
rules[10].rhs(), type(rules[10].rhs()[0])
```

Out[11]:

```
(('61',), str)
```

In [12]:

```python
rules[0].lhs(), type(rules[0].lhs())
```

Out[12]:

```
(S, nltk.grammar.Nonterminal)
```

In [13]:

```python
print(cnf_train[0].leaves())
```

```
['pierre', 'vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board', 'as', 'a', 'n
onexecutive', 'director', 'nov.', '29', '.']
```

ANSWER HERE

- productions(): Returns the list of CFG rules that "explain" the tree. For each non-terminal node in the tree, tree.productions() will return a production with the parent node as LHS and the children as RHS.
- rhs(): Returns the right-hand side of a production - its children nodes
- lhs(): Returns the left-hand side of a production - its parent node.
- leaves(): Returns the leaves of a production - meaning the sentence formed using the set of rules

Question 2

To count the number of unique rules, nonterminals and terminals, pleaseimplement functions **collect_rules, collect_nonterminals, collect_terminals**

In [14]:

```
def collect_rules(train_data):
    '''
    Collect the rules that appear in data.
    params:
        train_data: list[Tree] --- list of Tree objects
    return:
        rules: list[nltk.grammar.Production] --- list of rules (Production objects)
        rules_counts: Counter object --- a dictionary that maps one rule (nltk.Nonterminal) to its
number of
                                                occurences (int) in train data.
    '''
    rules = list()
    rules_counts = Counter()
    ### YOUR CODE HERE (~ 2 lines)

    # print(train_data[0].productions()
    [[rules.append(prod) for prod in rule.productions()] for rule in train_data]
    rules_counts = Counter(rules)
    # print(rules_counts.most_common(5))





    ### YOUR CODE HERE
    return rules, rules_counts


def collect_nonterminals(rules):
    '''
    collect nonterminals that appear in the rules
    params:
        rules: list[nltk.grammar.Production] --- list of rules (Production objects)
    return:
        nonterminals: set(nltk.Nonterminal) --- set of nonterminals
    '''
    nonterminals = list()
    ### YOUR CODE HERE (at least one line)


    for rule in rules:
      nonterminals.append(rule.lhs())
      for nonterm in rule.rhs():
        if(isinstance(nonterm,str)):
          continue
        else:
          nonterminals.append(nonterm)






    ### END OF YOUR CODE
    return set(nonterminals)

def collect_terminals(rules):
    '''
    collect terminals that appear in the rules
    params:
        rules: list[nltk.grammar.Production] --- list of rules (Production objects)
    return:
        terminals: set of strings --- set of terminals
    '''
    terminals = list()
    ### YOUR CODE HERE (at least one line)
    # [[terminals.append(strings) for strings in rule.leaves()] for rule in rules]
    # print(terminals[0])
    # print(rules[0])
    for rule in rules:
```

```
        for term in rule.rhs():
          if(isinstance(term,str)):
            terminals.append(term)
          else:
            continue




    ### END OF YOUR CODE

    return set(terminals)

# terminals = collect_terminals(train_rules)
```

In [15]:

```
train_rules, train_rules_counts = collect_rules(cnf_train)
nonterminals = collect_nonterminals(train_rules)
terminals = collect_terminals(train_rules)
```

In [16]:

```
### CORRECT ANSWER (19xxxx, 3xxxx, 1xxxx, 7xxx)
len(train_rules), len(set(train_rules)), len(terminals), len(nonterminals)
```

Out[16]:

```
(196646, 31656, 11367, 7869)
```

In [17]:

```
print(train_rules_counts.most_common(5))
```

```
[(, -> ',', 4876), (DT -> 'the', 4726), (. -> '.', 3814), (PP -> IN NP, 3273), (S|<VP-.> -> VP .,
3003)]
```

Question 3

Implement the function **build_pcfg** which builds a dictionary that stores the terminal rules and non-terminal rules.

In [19]:

```
def build_pcfg(rules_counts):
    '''
    Build a dictionary that stores the terminal rules and nonterminal rules.
    param:
        rules_counts: Counter object --- a dictionary that maps one rule to its number of
occurences in train data.
    return:
        rules_dict: dict(dict(dict)) --- a dictionary has a form like:
                    rules_dict = {'terminals':{'NP':{'the':1000,'an':500}, 'ADJ':
{'nice':500,'good':100}},
                                  'nonterminals':{'S':{'NP@VP':1000},'NP':{'NP@NP':540}}}
    When building "rules_dict", you need to use "lhs()", "rhs()" funtion and convert Nonterminal t
o str.
    All the keys in the dictionary are of type str.
    '@' is used as a special symbol to split left and right nonterminal strings.
    '''

    rules_dict = dict()
    ### rules_dict['terminals'] contains rules like "NP->'the'"
    ### rules_dict['nonterminals'] contains rules like "S->NP@VP"
    rules_dict['terminals'] = defaultdict(dict)
    rules_dict['nonterminals'] = defaultdict(dict)

    ### YOUR CODE HERE

    for rule in rules_counts:
```

```
      s = '@'
      temp_list = []

      if len(rule.rhs())==1:
          if isinstance(rule.rhs()[0],str):
            rules_dict['terminals'][str(rule.lhs())][rule.rhs()[0]] = rules_counts[rule]

      else:
        for child in rule.rhs():
          temp_list.append(str(child))
        s = s.join(temp_list)
        # rules_dict['nonterminals'][str(rule.lhs())] = defaultdict(dict)
        rules_dict['nonterminals'][str(rule.lhs())][s] = rules_counts[rule]






    ### END OF YOUR CODE
    return rules_dict
```

In [20]:

```
train_rules_dict = build_pcfg(train_rules_counts)
print(train_rules_dict['nonterminals']['ADJP'])
# print(train_rules_dict['nonterminals'].keys())
```

```
{'NP@JJ': 10, 'JJ@JJ': 8, 'JJ@S': 5, 'RB@JJ': 77, 'RB@VBN': 42, 'RB@JJR': 22, 'RB@DT': 2, 'JJ@PP':
42, 'CD@NN': 57, 'JJ@PRN': 1, 'QP@NN': 8, 'QP@-NONE-': 58, 'NP-ADV@JJR': 3, 'VBG@ADJP|<CC-VBG>': 1
, 'RB@ADJP|<JJ-CC-JJ>': 2, 'ADJP@PP': 9, 'RB@ADJP|<RB-JJ>': 3, 'VBN@PP': 2, 'QP@JJR': 1, 'RBR@JJ':
22, '$@ADJP|<JJ--NONE->': 7, 'ADVP-TMP+RB@VBN': 1, 'CD@NNS': 1, 'RB@ADJP|<JJ-PP>': 6, 'RB@ADJP|<RB
-JJ-PP>': 1, 'JJ@ADJP|<CD-NN>': 1, 'ADVP@JJ': 4, 'ADJP@ADJP|<,-ADJP-,-ADJP>': 1, 'ADVP-TMP@JJ': 1,
'RBS@JJ': 18, 'JJR@ADJP|<CC-JJR>': 2, '$@ADJP|<CD--NONE->': 28, 'ADVP-TMP+RB@ADJP|<RB-JJ>': 1, 'JJ
@NP-TMP': 10, 'ADVP@VBN': 2, 'JJ@ADJP|<CC-JJ>': 18, 'JJ@ADJP|<CC-NNP>': 1, 'JJ@SBAR': 2, 'JJ@PP-LO
C': 2, 'ADJP+JJ@ADJP|<,-CC-ADJP-,>': 1, 'NN@NN': 1, 'ADJP+JJ@ADJP|<CC-ADJP>': 1, 'RBS@ADJP|<RB-JJ>
': 1, 'NNP@JJ': 6, 'JJ@ADJP|<,-JJ-CC-JJ>': 3, 'JJR@JJ': 4, 'NNS@ADJP|<CC-NNS>': 1, 'NNP@ADJP|<,-
JJ>': 8, 'JJ@ADJP|<,-JJ-,-JJ-CC-NN>': 1, 'JJ@NP-TMP+CD': 11, 'NNP@NNP': 3, 'ADVP@ADJP|<JJ-PP>': 1,
"RB@ADJP|<``-JJ-CC-JJ-''>": 1, 'RB@RB': 4, 'JJ@ADJP|<,-CC-JJ>': 1, 'ADJP@ADJP|<CC-ADJP>': 1, 'RB@A
DJP|<VBN-PRT+RP-PP>': 1, 'JJS@JJ': 3, 'VBN@PP-CLR': 1, 'RB@ADJP|<JJ-PP-LOC>': 1, 'JJ@JJR': 1, 'JJ@
ADJP|<CC-VBG>': 1, 'JJ@ADJP|<RB-SBAR+-NONE->': 1, 'NN@ADJP|<RB-SBAR+-NONE->': 1, 'QP@ADJP|<RB-JJ>'
: 1, 'ADVP-TMP+RB@JJ': 1, 'ADJP@ADJP|<,-CC-ADJP>': 1, 'RB@ADJP|<RB-PP-S>': 1, 'RB@ADJP|<RB-S>': 1,
'NN@JJ': 1, 'ADJP@ADJP|<CC-ADJP+JJ>': 2, 'RB@ADJP|<RBR-JJ>': 2, 'NNS@ADJP|<PRN-VBN>': 1, 'RB@ADJP|
<JJR-IN>': 1, 'JJR@VBN': 1, 'RB@VBG': 2, 'RBS@VBN': 1, 'JJR@PP': 1, 'RBR@ADJP|<JJ-PP>': 1,
'ADVP+RB@ADJP|<RB-JJR>': 1, 'DT@ADJP|<ADJP+JJ-CC-ADJP+JJ>': 1, "``@ADJP|<JJ-''-CC-``-JJ>": 1,
'JJS@ADJP|<JJ-S>': 1, 'JJ@RB': 1, '$@ADJP|<CD-JJ>': 1, 'CD@ADJP|<CD-NN>': 8, 'JJ@PP-TMP': 2, 'JJ@J
JS': 1, 'VB@JJR': 1, 'NP-ADV@JJ': 1, 'JJ@NP': 1, 'VBN@JJ': 1, 'NNP@ADJP|<NNP-,-JJ>': 1,
'ADJP+JJR@PP': 1, 'NN@ADJP|<CC-NN>': 1, '$@JJ': 1, "``@ADJP|<RB-''-JJR>": 1, 'JJ@ADJP|<CC-RB>': 1,
'CD@JJ': 1, 'IN@NN': 1, 'NP@ADJP|<JJ-PP>': 1, '``@ADJP|<RB-VBN>': 1, 'NNP@ADJP|<,-NNP-JJ>': 1, 'AD
JP@ADJP|<CC-ADJP+JJR>': 1, 'JJ@VBN': 1, 'VBN@ADJP|<CC-JJ>': 1}
```

Question 4

Estimate the probability of rule $NP\rightarrow NNP@NNP$

In [21]:

```
np_count = 0
for val in train_rules_dict['nonterminals']['NP'].values():
  np_count += val
# print(np_count)
print("The probability of the rule is: " + str(train_rules_dict['nonterminals']['NP']['NNP@NNP']/n
p_count))
# prob = train_rules_dict['nonterminals'][NP]
```

```
The probability of the rule is: 0.03950843529348353
```

Question 5

Find the terminal symbols in "cnf_test[0]" that never appeared in the PCFG we built.

In [22]:

```
# print(cnf_test[0].leaves())
# for dic in train_rules_dict['terminals'].values():
ret = cnf_test[0].leaves()
for terminal in cnf_test[0].leaves():
  for dic in train_rules_dict['terminals'].values():
    # print(dic)
    if terminal in dic:
      try:
        ret.remove(terminal)
      except ValueError:
        pass

print(ret)
```

```
['constitutional-law']
```

Question 6

We can use smoothing techniques to handle these cases. A simple smoothing method is as follows. We first create a new "unknown" terminal symbol unk.

Next, for each original non-terminal symbol $A\in N$, we add one new rule $A \rightarrow unk$ to the original PCFG.

The smoothed probabilities for all rules can then be estimated as: $$q_{smooth}(A \rightarrow \beta) = \frac {count(A \rightarrow \beta)}{count(A)+1}$$ $$q_{smooth}(A \rightarrow unk) = \frac {1}{count(A)+1}$$ where $|V|$ is the count of unique terminal symbols.

Implement the function **smooth_rules_prob** which returns the smoothed rule probabilities

In [24]:

```
def smooth_rules_prob(rules_counts):
    '''
    params:
        rules_counts: dict(dict(dict)) --- a dictionary has a form like:
                      rules_counts = {'terminals':{'NP':{'the':1000,'an':500}, 'ADJ':
{'nice':500,'good':100}},
                                      'nonterminals':{'S':{'NP@VP':1000},'NP':{'NP@NP':540}}}

    return:
        rules_prob: dict(dict(dict)) --- a dictionary that has a form like:
                                 rules_prob = {'terminals':{'NP':{'the':0.6,'an':0.3, '<unk>':0.1},
                                                            'ADJ':{'nice':0.6,'good':0.3,'<unk>':0.1
                                                            'S':{'<unk>':0.01}}}
                                         'nonterminals':{'S':{'NP@VP':0.99}}
    '''
    rules_prob = copy.deepcopy(rules_counts)
    unk = '<unk>'
    ### Hint: don't forget to consider nonterminal symbols that don't appear in
rules_counts['terminals'].keys()
    ### YOUR CODE HERE

    for dic in rules_prob['terminals'].values():
      total = 0
      for val in dic.values():
        total += val
      for key,val in dic.items():
        # print(key,val)
        dic[key] = val/(total+1)
      dic[unk] = 1/(total+1)
      # print(dic[unk])

    for key,dic in rules_prob['nonterminals'].items():
      total = 0
      for val in dic.values():
        total += val
      for k,val in dic.items():
        # print(key,val)
        dic[k] = val/(total+1)
      if key not in rules_prob['terminals']:
```

```
        rules_prob['terminals'][key] = {unk:(1/(total+1))}









    ### END OF YOUR CODE
    return rules_prob
```

In [25]:

```
s_rules_prob = smooth_rules_prob(train_rules_dict)
terminals.add('<unk>')
```

In [26]:

```
print(s_rules_prob['nonterminals']['S']['NP-SBJ@S|<VP-.>'])
print(s_rules_prob['nonterminals']['S']['NP-SBJ-1@S|<VP-.>'])
print(s_rules_prob['nonterminals']['NP']['NNP@NNP'])
print(s_rules_prob['terminals']['NP'])
```

```
0.1300172371337109
0.025240088648116228
0.039506305917861376
{'<unk>': 5.389673385792821e-05}
```

In [27]:

```
len(terminals)
```

Out[27]:

```
11368
```

CKY Algorithm

Similar to the Viterbi algorithm, the CKY algorithm is a dynamic-programming algorithm. Given a PCFG $G=(N, \ \Sigma, \ S, \ R, \ q)$, we can use the CKY algorithm described in class to find the highest scoring parse tree for a sentence.

First, let us complete the *CKY* function from scratch using only Python built-in functions and the Numpy package.

The output should be two dictionaries π and bp, which store the optimal probability and backpointer information respectively.

Given a sentence $w_0, w_1, ...,w_{n-1}$, $\pi(i, k, X)$, $bp(i, k, X)$ refer to the highest score and backpointer for the (partial) parse tree that has the root X (a non-terminal symbol) and covers the word span $w_i, ..., w_{k-1}$, where $0 \le i < k \le n$. Note that a backpointer includes both the best grammar rule chosen and the best split point.

Question 7

Implement **CKY** function and run the test code to check your implementation.

In [28]:

```
import math
def CKY(sent, rules_prob):
    '''
    params:
        sent: list[str] --- a list of strings
        rules_prob: dict(dict(dict)) --- a dictionary that has a form like:
                                  rules_prob = {'terminals':{'NP':{'the':0.6,'an':0.3, '<u
k>':0.1},
                                                           'ADJ':
{'nice':0.6,'good':0.3,'<unk>':0.1},
```

```
{'nice':0.6,'good':0.3,'<unk>':0.1},
                                                                        'S':{'<unk>':0.01}}}
                                                           'nonterminals':{'S':{'NP@VP':0.99}}
    return:
        score: dict() --- score[(i,i+span)][root] represents the highest score for the parse
(sub)tree that has the root "root"
                            across words w_i, w_{i+1},..., w_{i+span-1}.
        back: dict() --- back[(i,i+span)][root] = (split , left_child, right_child); split: int;
                          left_child: str; right_child: str.
    '''
    score = defaultdict(dict)
    back = defaultdict(dict)
    sent_len = len(sent)
    ### YOUR CODE HERE


    word_set = set()
    for term in rules_prob['terminals'].keys():
        for word in rules_prob['terminals'][term].keys():
            word_set.add(word)

    split = 0
    for i in range(sent_len):
      for term_key,term_val in rules_prob['terminals'].items():
        if sent[i] in word_set and sent[i] in rules_prob['terminals'][term_key]:
            score[(i,i+1)][term_key] = math.log(rules_prob['terminals'][term_key][sent[i]])
            back[(i,i+1)][term_key] = (split,sent[i],None)
        elif sent[i] in word_set and sent[i] not in rules_prob['terminals'][term_key]:
          continue
        else:
          score[(i,i+1)][term_key] = math.log(rules_prob['terminals'][term_key]['<unk>'])
          back[(i,i+1)][term_key] = (split,sent[i],None)


    for span in range(2,sent_len+1):
      for begin in range(0,sent_len-span+1):
        end=begin+span
        for split in range(begin+1,end):
          for nonterm_key in rules_prob['nonterminals'].keys():
            for combined_child, probability in rules_prob['nonterminals'][nonterm_key].items():

              left_child, right_child = combined_child.split('@')
              left_childs = back[(begin,split)]
              right_childs = back[(split,end)]

              if left_child in left_childs and right_child in right_childs:

                if nonterm_key not in score[(begin,end)]:
                  score[(begin,end)][nonterm_key] = math.log(probability) + score[(begin,split)]
[left_child]+ score[(split,end)][right_child]
                  back[(begin,end)][nonterm_key] = (split,left_child,right_child)
                else:
                  temp = math.log(probability) + score[(begin,split)][left_child]+ score[(split,end
)][right_child]
                  if temp > score[(begin,end)][nonterm_key]:
                      score[(begin,end)][nonterm_key] = temp
                      back[(begin,end)][nonterm_key] = (split,left_child,right_child)









    ### END OF YOUR CODE
    return score, back

sent = cnf_train[0].leaves()
score, back = CKY(sent, s_rules_prob)
print(score[(0, len(sent))]['S'])
```

```
-117.52227496068694
```

In [29]:

```
sent = cnf_train[0].leaves()
score, back = CKY(sent, s_rules_prob)
```

In [30]:

```
score[(0, len(sent))]['S']
```

Out[30]:

```
-117.52227496068694
```

Question 8

Implement **build_tree** function according to algorithm 2 to reconstruct theparse tree

In [31]:

```
def build_tree(back, root,nonterminals):
    '''
    Build the tree recursively.
    params:
        back: dict() --- back[(i,i+span)][X] = (split , left_child, right_child); split:int; left_child: str; right_child: str.
        root: tuple() --- (begin, end, nonterminal_symbol), e.g., (0, 10, 'S
    return:
        tree: nltk.tree.Tree
    '''
    begin = root[0]
    end = root[1]
    root_label = root[2]
    ### YOUR CODE HERE
    split, left_label, right_label = back[(begin, end)][root_label]
    if split == 0:
        tree = Tree(root_label, [left_label])
    else:
        left_child = build_tree(back, (begin, split, left_label), nonterminals)
        right_child = build_tree(back, (split, end, right_label), nonterminals)
        tree = Tree(root_label, [left_child, right_child])
    ### END OF YOUR CODE
    return tree
```

In [32]:

```
build_tree(back, (0, len(sent), 'S'), nonterminals).pprint()
```

```
(S
  (NP-SBJ
    (NP (NNP pierre) (NNP vinken))
    (NP-SBJ|<,-NP-,>
      (, ,)
      (NP-SBJ|<NP-,>
        (NP (CD 61) (NP|<NNS-JJ> (NNS years) (JJ old)))
        (, ,))))
  (S|<VP-.>
    (VP
      (MD will)
      (VP
        (VB join)
        (VP|<NP-PP-CLR-NP-TMP>
          (NP (DT the) (NN board))
          (VP|<PP-CLR-NP-TMP>
            (PP-CLR
              (IN as)
              (NP
                (DT a)
                (NP|<JJ-NN> (JJ nonexecutive) (NN director))))
            (NP-TMP (NNP nov.) (CD 29))))))
    (. .)))
```

Question 9

In [33]:

```
def set_leave_index(tree):
    '''
    Label the leaves of the tree with indexes
    Arg:
        tree: original tree, nltk.tree.Tree
    Return:
        tree: preprocessed tree, nltk.tree.Tree
    '''
    for idx, _ in enumerate(tree.leaves()):
        tree_location = tree.leaf_treeposition(idx)
        non_terminal = tree[tree_location[:-1]]
        non_terminal[0] = non_terminal[0] + "_" + str(idx)
    return tree

def get_nonterminal_bracket(tree):
    '''
    Obtain the constituent brackets of a tree
    Arg:
        tree: tree, nltk.tree.Tree
    Return:
        nonterminal_brackets: constituent brackets, set
    '''
    nonterminal_brackets = set()
    for tr in tree.subtrees():
        label = tr.label()
        #print(tr.leaves())
        if len(tr.leaves()) == 0:
            continue
        start = tr.leaves()[0].split('_')[-1]
        end = tr.leaves()[-1].split('_')[-1]
        if start != end:
            nonterminal_brackets.add(label+'-('+start+':'+end+')')
    return nonterminal_brackets

def word2lower(w, terminals):
    '''
    Map an unknow word to "unk"
    '''
    return w.lower() if w in terminals else '<unk>'
```

In [34]:

```
correct_count = 0
pred_count = 0
gold_count = 0
for i, t in enumerate(cnf_test):
    #Protect the original tree
    t = copy.deepcopy(t)
    sent = t.leaves()
    #Map the unknow words to "unk"
    sent = [word2lower(w.lower(), terminals) for w in sent]

    #CKY algorithm
    score, back = CKY(sent, s_rules_prob)
    candidate_tree = build_tree(back, (0, len(sent), 'S'), nonterminals)

    #Extract constituents from the gold tree and predicted tree
    pred_tree = set_leave_index(candidate_tree)
    pred_brackets = get_nonterminal_bracket(pred_tree)

    #Count correct constituents
    pred_count += len(pred_brackets)
    gold_tree = set_leave_index(t)
    gold_brackets = get_nonterminal_bracket(gold_tree)
    gold_count += len(gold_brackets)
    current_correct_num = len(pred_brackets.intersection(gold_brackets))
    correct_count += current_correct_num

    print('#'*20)
    print('Test Tree:', i+1)
    print('Constituent number in the predicted tree:', len(pred_brackets))
```

```
    print('Constituent number in the predicted tree:', len(pred_brackets))
    print('Constituent number in the gold tree:', len(gold_brackets))
    print('Correct constituent number:', current_correct_num)

recall = correct_count/gold_count
precision = correct_count/pred_count
f1 = 2*recall*precision/(recall+precision)
```

```
####################
Test Tree: 1
Constituent number in the predicted tree: 20
Constituent number in the gold tree: 20
Correct constituent number: 14
####################
Test Tree: 2
Constituent number in the predicted tree: 54
Constituent number in the gold tree: 54
Correct constituent number: 28
####################
Test Tree: 3
Constituent number in the predicted tree: 30
Constituent number in the gold tree: 30
Correct constituent number: 23
####################
Test Tree: 4
Constituent number in the predicted tree: 17
Constituent number in the gold tree: 17
Correct constituent number: 16
####################
Test Tree: 5
Constituent number in the predicted tree: 32
Constituent number in the gold tree: 32
Correct constituent number: 26
####################
Test Tree: 6
Constituent number in the predicted tree: 40
Constituent number in the gold tree: 40
Correct constituent number: 18
####################
Test Tree: 7
Constituent number in the predicted tree: 22
Constituent number in the gold tree: 22
Correct constituent number: 7
####################
Test Tree: 8
Constituent number in the predicted tree: 18
Constituent number in the gold tree: 18
Correct constituent number: 6
####################
Test Tree: 9
Constituent number in the predicted tree: 28
Constituent number in the gold tree: 28
Correct constituent number: 16
####################
Test Tree: 10
Constituent number in the predicted tree: 40
Constituent number in the gold tree: 40
Correct constituent number: 8
```

In [35]:

```
print('Overall precision: {:.3f}, recall: {:.3f}, f1: {:.3f}'.format(precision, recall, f1))
```

```
Overall precision: 0.538, recall: 0.538, f1: 0.538
```

In [36]:

```
print('Overall precision: {:.3f}, recall: {:.3f}, f1: {:.3f}'.format(precision, recall, f1))
```

```
Overall precision: 0.538, recall: 0.538, f1: 0.538
```

In [37]:

```
et=time.time()
print(et - st)
```

```
874.4197461605072
```

In [ ]:

```

```