In [1]:
```
import os
os.environ['CLASSPATH'] = 'C:/Program Files/stanford-parser-full-2020-11-17'
```

In [8]:
```
# constituency_parsing.py
sentence = 'the quick red riding food jumped over the small log'

from nltk.parse.stanford import StanfordParser

scp = StanfordParser(model_path='C:/Program Files/stanford-parser-full-2020-11-17/edu/



result = list(scp.raw_parse(sentence))
print(result[0])

result[0].draw()

import nltk
from nltk.grammar import Nonterminal
from nltk.corpus import treebank

training_set = treebank.parsed_sents()

print(training_set[1])

# extract the productions for all annotated training sentences
treebank_productions = list(
                        set(production
                            for sent in training_set
                            for production in sent.productions()
                        )
                    )

treebank_productions[0:10]

# add productions for each word, POS tag
for word, tag in treebank.tagged_words():
        t = nltk.Tree.fromstring("("+ tag + " " + word  +")")
        for production in t.productions():
                treebank_productions.append(production)

# build the PCFG based grammar
treebank_grammar = nltk.grammar.induce_pcfg(Nonterminal('S'),
                                            treebank_productions)

# build the parser
viterbi_parser = nltk.ViterbiParser(treebank_grammar)

# get sample sentence tokens
tokens = nltk.word_tokenize(sentence)

# get parse tree for sample sentence
result = list(viterbi_parser.parse(tokens))


# get tokens and their POS tags from pattern package
from pattern.en import tag as pos_tagger
tagged_sent = pos_tagger(sentence)
```

```
# use NLTK POS tagger instead
tokens = nltk.word_tokenize(sentence)
tagged_sent = nltk.pos_tag(tokens)

print(tagged_sent)

# extend productions for sample sentence tokens
for word, tag in tagged_sent:
    t = nltk.Tree.fromstring("("+ tag + " " + word  +")")
    for production in t.productions():
        treebank_productions.append(production)

# rebuild grammar
treebank_grammar = nltk.grammar.induce_pcfg(Nonterminal('S'),
                                            treebank_productions)

# rebuild parser
viterbi_parser = nltk.ViterbiParser(treebank_grammar)

# get parse tree for sample sentence
result = list(viterbi_parser.parse(tokens))

print(result[0])
result[0].draw()
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_11252\285918331.py:13: DeprecationWarning:
The StanfordParser will be deprecated
Please use nltk.parse.corenlp.CoreNLPParser instead.
  scp = StanfordParser(model_path='C:/Program Files/stanford-parser-full-2020-11-17/e
du/stanford/nlp/models/lexparser/englishPCFG.ser.gz')
```

```
(ROOT
  (S
    (NP (DT the) (JJ quick) (JJ red) (NN riding) (NN food))
    (VP
      (VBD jumped)
      (PP (IN over) (NP (DT the) (JJ small) (NN log))))))
(S
  (NP-SBJ (NNP Mr.) (NNP Vinken))
  (VP
    (VBZ is)
    (NP-PRD
      (NP (NN chairman))
      (PP
        (IN of)
        (NP
          (NP (NNP Elsevier) (NNP N.V.))
          (, ,)
          (NP (DT the) (NNP Dutch) (VBG publishing) (NN group))))))
  (. .))
[('the', 'DT'), ('quick', 'JJ'), ('red', 'JJ'), ('riding', 'NN'), ('food', 'NN'), ('j
umped', 'VBD'), ('over', 'IN'), ('the', 'DT'), ('small', 'JJ'), ('log', 'NN')]
(S
  (NP-SBJ
    (DT the)
    (JJ quick)
    (JJ red)
    (NN riding)
    (NN food))
  (ADVP-PRD
    (VBN jumped)
    (PP (IN over) (NP-LGS (DT the) (JJ small) (NN log)))))) (p=7.38607e-37)
```

In [3]:

```
# pos_tagging.py
sentence = "I saw the man with the telescope but he didn't see me"


# Using NLTK's built-in tagger based on PTB
import nltk
tokens = nltk.word_tokenize(sentence)
tagged_sent = nltk.pos_tag(tokens, tagset='universal')
print(tagged_sent)


# Using the pattern package (Python 2.x only) built-in tagger (optional)
# from pattern.en import tag
# tagged_sent = tag(sentence)
# print(tagged_sent)



# Building your own tagger
# - default tagger that tags all words the same!
# - regex tagger that doesn't care about context (most common tag per word)

# Fortunately the treebank corpus is bundled with NLTK for training a tagger
# We need to divide the data into training and test sets first
from nltk.corpus import treebank
data = treebank.tagged_sents()
train_data = data[:3500]
test_data = data[3500:]
```

```
print(train_data[0])

# SAQ 1: How much data is there for training, testing?
# SAQ 2: What is the last training sentence; test sentence?

# Default 'naive' tagger - tags all words with a given tag!
from nltk.tag import DefaultTagger
dt = DefaultTagger('NN') # Can specify any default tag - NN gives best score - why?

# Test score and example sentence tag output
print(dt.evaluate(test_data))
print(dt.tag(tokens))


# Regex tagger
from nltk.tag import RegexpTagger
# Define 'fixed' regex tag patterns
patterns = [
        (r'.*ing$', 'VBG'),               # gerunds
        (r'.*ed$', 'VBD'),                # simple past
        (r'.*es$', 'VBZ'),                # 3rd singular present
        (r'.*ould$', 'MD'),               # modals
        (r'.*\'s$', 'NN$'),               # possessive nouns
        (r'.*s$', 'NNS'),                 # plural nouns
        (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # cardinal numbers
        (r'.*', 'NN')                     # nouns (default) ...
]
rt = RegexpTagger(patterns)

# Test score and example sentence tag output
print(rt.evaluate(test_data))
print(rt.tag(tokens))


# Training your own tagger
# 1. using n-gram taggers and combining them with backoff
# 2. using naive bayes (statistical) model
# 3. using maximum entropy (classifier) model

## N gram taggers
from nltk.tag import UnigramTagger # Context insentitive
from nltk.tag import BigramTagger  # Considers previous word
from nltk.tag import TrigramTagger # Considers previous 2 words

# Traing the taggers
ut = UnigramTagger(train_data)
bt = BigramTagger(train_data)
tt = TrigramTagger(train_data)

# Test UnigramTagger score and example sentence tag output
print(ut.evaluate(test_data))
print(ut.tag(tokens))

# Test BigramTagger score and example sentence tag output
print(bt.evaluate(test_data))
print(bt.tag(tokens))

# Test TrigramTagger score and example sentence tag output
print(tt.evaluate(test_data))
print(tt.tag(tokens))
```

```
# Combining all 3 n-gram taggers with backoff (smoothing)
def combined_tagger(train_data, taggers, backoff=None):
    for tagger in taggers:
        backoff = tagger(train_data, backoff=backoff)
    return backoff

ct = combined_tagger(train_data=train_data,
                     taggers=[UnigramTagger, BigramTagger, TrigramTagger],
                     backoff=rt)

# Test Combined n-gram tagger score and example sentence tag output
print(ct.evaluate(test_data))
print(ct.tag(tokens))



# Treating POS tagging as a classification problem
# We use the ClassifierBasedPOSTagger class to build a classifier by specifying some
# classification algorithm - here the NaiveBayes abd Maxent algorithms which are passe
# to the class via the classifier_builder parameter
from nltk.classify import NaiveBayesClassifier, MaxentClassifier
from nltk.tag.sequential import ClassifierBasedPOSTagger

# First a Naive Bayes (statistical) classifier
nbt = ClassifierBasedPOSTagger(train=train_data,
                               classifier_builder=NaiveBayesClassifier.train)

# Test NBC tagger score and example sentence tag output
print(nbt.evaluate(test_data))
print(nbt.tag(tokens))


# Finally a Maximum entropy classifier (that would take sometime)
# met = ClassifierBasedPOSTagger(train=train_data,
#                                classifier_builder=MaxentClassifier.train)

met = ClassifierBasedPOSTagger(train=train_data,
                               classifier_builder=lambda train_feats: MaxentClassifier

# Test Maxent tagger score and example sentence tag output
print(met.evaluate(test_data))
print(met.tag(tokens))


# Final accuracies
print('Tagger accuracies:')
print()
print('Default tagger %.2f' %dt.evaluate(test_data))
print('Regex tagger %.2f' %rt.evaluate(test_data))
print('Unigram tagger %.2f' %ut.evaluate(test_data))
print('Bigram tagger %.2f' %bt.evaluate(test_data))
print('Trigram tagger %.2f' %tt.evaluate(test_data))
print('Combined tagger %.2f' %ct.evaluate(test_data))
print('Naive Bayes tagger %.2f' %nbt.evaluate(test_data))
print('Maxent tagger %.2f' %met.evaluate(test_data))
```

```
[('I', 'PRON'), ('saw', 'VERB'), ('the', 'DET'), ('man', 'NOUN'), ('with', 'ADP'),
('the', 'DET'), ('telescope', 'NOUN'), ('but', 'CONJ'), ('he', 'PRON'), ('did', 'VER
B'), ("n't", 'ADV'), ('see', 'VERB'), ('me', 'PRON')]
[('Pierre', 'NNP'), ('Vinken', 'NNP'), (',', ','), ('61', 'CD'), ('years', 'NNS'),
('old', 'JJ'), (',', ','), ('will', 'MD'), ('join', 'VB'), ('the', 'DT'), ('board',
'NN'), ('as', 'IN'), ('a', 'DT'), ('nonexecutive', 'JJ'), ('director', 'NN'), ('No
v.', 'NNP'), ('29', 'CD'), ('.', '.')]
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:39: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(dt.evaluate(test_data))
0.1454158195372253
[('I', 'NN'), ('saw', 'NN'), ('the', 'NN'), ('man', 'NN'), ('with', 'NN'), ('the', 'N
N'), ('telescope', 'NN'), ('but', 'NN'), ('he', 'NN'), ('did', 'NN'), ("n't", 'NN'),
('see', 'NN'), ('me', 'NN')]
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:59: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(rt.evaluate(test_data))
0.24039113176493368
[('I', 'NN'), ('saw', 'NN'), ('the', 'NN'), ('man', 'NN'), ('with', 'NN'), ('the', 'N
N'), ('telescope', 'NN'), ('but', 'NN'), ('he', 'NN'), ('did', 'NN'), ("n't", 'NN'),
('see', 'NN'), ('me', 'NN')]
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:79: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(ut.evaluate(test_data))
0.8607803272340013
[('I', 'PRP'), ('saw', 'VBD'), ('the', 'DT'), ('man', 'NN'), ('with', 'IN'), ('the',
'DT'), ('telescope', None), ('but', 'CC'), ('he', 'PRP'), ('did', 'VBD'), ("n't", 'R
B'), ('see', 'VB'), ('me', 'PRP')]
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:83: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(bt.evaluate(test_data))
0.13466937748087907
[('I', 'PRP'), ('saw', None), ('the', None), ('man', None), ('with', None), ('the', N
one), ('telescope', None), ('but', None), ('he', None), ('did', None), ("n't", None),
('see', None), ('me', None)]
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:87: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(tt.evaluate(test_data))
0.08064672281924679
[('I', 'PRP'), ('saw', None), ('the', None), ('man', None), ('with', None), ('the', N
one), ('telescope', None), ('but', None), ('he', None), ('did', None), ("n't", None),
('see', None), ('me', None)]
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:101: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(ct.evaluate(test_data))
```

```
0.9094781682641108
[('I', 'PRP'), ('saw', 'VBD'), ('the', 'DT'), ('man', 'NN'), ('with', 'IN'), ('the',
'DT'), ('telescope', 'NN'), ('but', 'CC'), ('he', 'PRP'), ('did', 'VBD'), ("n't", 'R
B'), ('see', 'VB'), ('me', 'PRP')]
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:118: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(nbt.evaluate(test_data))
0.9306806079969019
[('I', 'PRP'), ('saw', 'VBD'), ('the', 'DT'), ('man', 'NN'), ('with', 'IN'), ('the',
'DT'), ('telescope', 'NN'), ('but', 'CC'), ('he', 'PRP'), ('did', 'VBD'), ("n't", 'R
B'), ('see', 'VB'), ('me', 'PRP')]
  ==> Training (10 iterations)

      Iteration    Log Likelihood    Accuracy
      ---------------------------------------
             1          -3.82864        0.007
             2          -0.76176        0.957
C:\Users\User\AppData\Roaming\Python\Python310\site-packages\nltk\classify\maxent.py:
1381: RuntimeWarning: overflow encountered in power
  exp_nf_delta = 2**nf_delta
C:\Users\User\AppData\Roaming\Python\Python310\site-packages\nltk\classify\maxent.py:
1383: RuntimeWarning: invalid value encountered in multiply
  sum1 = numpy.sum(exp_nf_delta * A, axis=0)
C:\Users\User\AppData\Roaming\Python\Python310\site-packages\nltk\classify\maxent.py:
1384: RuntimeWarning: invalid value encountered in multiply
  sum2 = numpy.sum(nf_exp_nf_delta * A, axis=0)
         Final               nan        0.984
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:130: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(met.evaluate(test_data))
0.9270016458514861
[('I', 'PRP'), ('saw', 'VBD'), ('the', 'DT'), ('man', 'NN'), ('with', 'IN'), ('the',
'DT'), ('telescope', 'NN'), ('but', 'CC'), ('he', 'PRP'), ('did', 'VBD'), ("n't", 'R
B'), ('see', 'VB'), ('me', 'PRP')]
Tagger accuracies:

C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:137: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print('Default tagger %.2f' %dt.evaluate(test_data))
Default tagger 0.15
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:138: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print('Regex tagger %.2f' %rt.evaluate(test_data))
Regex tagger 0.24
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:139: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print('Unigram tagger %.2f' %ut.evaluate(test_data))
Unigram tagger 0.86
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:140: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print('Bigram tagger %.2f' %bt.evaluate(test_data))
```

```
Bigram tagger 0.13
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:141: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print('Trigram tagger %.2f' %tt.evaluate(test_data))
```

```
Trigram tagger 0.08
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:142: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print('Combined tagger %.2f' %ct.evaluate(test_data))
```

```
Combined tagger 0.91
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:143: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print('Naive Bayes tagger %.2f' %nbt.evaluate(test_data))
```

```
Naive Bayes tagger 0.93
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_11252\3240707400.py:144: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print('Maxent tagger %.2f' %met.evaluate(test_data))
```

```
Maxent tagger 0.93
```

In [4]:

```
#shallow_parsing.py

# Training your own chunker using chunked treeband data - again made available in NLTK
# As before (with tagging) we first divide the data into training and testing sets
from nltk.corpus import treebank_chunk
data = treebank_chunk.chunked_sents()
train_data = data[0:3500]
test_data = data[3500:]
print(train_data[0])

simple_sentence = 'the brown fox jumped over the lazy dog'

# Can use tagger from package pattern.en if using Python 2.x
# from pattern.en import tag
# tagged_sentence = tag(sentence)

import nltk
from nltk.chunk import RegexpParser
tokens = nltk.word_tokenize(simple_sentence)
tagged_simple_sent = nltk.pos_tag(tokens)
print(tagged_simple_sent)

# We first define our grammars using regex pattern using the RegexpParser
# We can specify which patterns we want to segment in a sentence as *chunks*
chunk_grammar = """
NP: {<DT>?<JJ>*<NN.*>}
VP: {<VBD><IN>}
```

```
"""
rc = RegexpParser(chunk_grammar)
c = rc.parse(tagged_simple_sent)
print(c)

# We sometimes want to specify which patterns we DO NOT want to segment in a sentence
# so that we can *chunk* all the others
chink_grammar = """
NP: {<JJ|NN>+} # chunk only adjective-noun pair as NP
"""

rc = RegexpParser(chink_grammar)
c = rc.parse(tagged_simple_sent)
print(c)


# A more realistic grammar for chunking
grammar = """
NP: {<DT>?<JJ>?<NN.*>}
ADJP: {<JJ>}
ADVP: {<RB.*>}
PP: {<IN>}
VP: {<MD>?<VB.*>+}

"""

# And a more realistic sentence as input
sentence = 'the brown fox is quick and he may jump over the lazy dog'
tokens = nltk.word_tokenize(sentence)
tagged_sent = nltk.pos_tag(tokens)

rc = RegexpParser(grammar)
c = rc.parse(tagged_sent)
print(c)

print(rc.evaluate(test_data))
# The performance is not great!
# Why is this?


# We have acecss to a utility function tree2conlltags which extracts word, tag and
# chunk triples from annotated text
from nltk.chunk.util import tree2conlltags, conlltags2tree
# Let's take a slightly more typical sentence from our data
train_sent = train_data[7]
print(train_sent)

# We extract the POS and chunk tags using tree2conlltags function which returns a list
wtc = tree2conlltags(train_sent)
wtc

# We can 'reverse' this to output a shallow tree using the conlltags2tree function
tree = conlltags2tree(wtc)
print(tree)


# We can use these features to train a 'combined' chunker as we did for POS tagging
def conll_tag_chunks(chunk_sents):
  tagged_sents = [tree2conlltags(tree) for tree in chunk_sents]
  return [[(t, c) for (w, t, c) in sent] for sent in tagged_sents]
```

```
def combined_tagger(train_data, taggers, backoff=None):
    for tagger in taggers:
        backoff = tagger(train_data, backoff=backoff)
    return backoff

from nltk.tag import UnigramTagger, BigramTagger
from nltk.chunk import ChunkParserI

# We create a new class to use the word, POS and Chunk tag features to train a chunker
# that is able to 'backoff' from bigram to a unigram model as before
# Can you have another layer for trigram and back off to this model?
class NGramTagChunker(ChunkParserI):

  def __init__(self, train_sentences,
               tagger_classes=[UnigramTagger, BigramTagger]):
    train_sent_tags = conll_tag_chunks(train_sentences)
    self.chunk_tagger = combined_tagger(train_sent_tags, tagger_classes)

  def parse(self, tagged_sentence):
    if not tagged_sentence:
        return None
    pos_tags = [tag for word, tag in tagged_sentence]
    chunk_pos_tags = self.chunk_tagger.tag(pos_tags)
    chunk_tags = [chunk_tag for (pos_tag, chunk_tag) in chunk_pos_tags]
    wpc_tags = [(word, pos_tag, chunk_tag) for ((word, pos_tag), chunk_tag)
                     in zip(tagged_sentence, chunk_tags)]
    return conlltags2tree(wpc_tags)

# We call our new class and pass it the training data from the chunked treebank
ntc = NGramTagChunker(train_data)
print(ntc.evaluate(test_data))
# Now we get really good results on the data set
# Why?

# Let's try to visualize the chunk for the more realistic sample sentence
tree = ntc.parse(tagged_sent)
print(tree)
tree.draw()


# We now use our shallow parser on a larger 'Wall Street Journal' corpus
# SAQ 1. How big is it?
# SAQ 2. How much test data did we have before, and how much now?
from nltk.corpus import conll2000
wsj_data = conll2000.chunked_sents()
train_wsj_data = wsj_data[:7500]
test_wsj_data = wsj_data[7500:]
print(train_wsj_data[10])

# We first train our model on the training corpus
tc = NGramTagChunker(train_wsj_data)
# And then we test it on the test data
print(tc.evaluate(test_wsj_data))
```

```
(S
  (NP Pierre/NNP Vinken/NNP)
  ,/,
  (NP 61/CD years/NNS)
  old/JJ
  ,/,
  will/MD
  join/VB
  (NP the/DT board/NN)
  as/IN
  (NP a/DT nonexecutive/JJ director/NN Nov./NNP 29/CD)
  ./.)
[('the', 'DT'), ('brown', 'JJ'), ('fox', 'NN'), ('jumped', 'VBD'), ('over', 'IN'),
('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]
(S
  (NP the/DT brown/JJ fox/NN)
  (VP jumped/VBD over/IN)
  (NP the/DT lazy/JJ dog/NN))
(S
  the/DT
  (NP brown/JJ fox/NN)
  jumped/VBD
  over/IN
  the/DT
  (NP lazy/JJ dog/NN))
(S
  (NP the/DT brown/JJ fox/NN)
  (VP is/VBZ)
  (ADJP quick/JJ)
  and/CC
  he/PRP
  (VP may/MD jump/VB)
  (PP over/IN)
  (NP the/DT lazy/JJ dog/NN))
C:\Users\User\AppData\Local\Temp\ipykernel_11252\1847143938.py:63: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(rc.evaluate(test_data))
```

```
ChunkParse score:
    IOB Accuracy:  46.1%%
    Precision:     19.9%%
    Recall:        43.3%%
    F-Measure:     27.3%%
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
C:\Users\User\AppData\Local\Temp\ipykernel_11252\1847143938.py:119: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(ntc.evaluate(test_data))
ChunkParse score:
    IOB Accuracy:  97.2%%
    Precision:     91.4%%
    Recall:        94.3%%
    F-Measure:     92.8%%
(S
  (NP the/DT brown/JJ fox/NN)
  is/VBZ
  (NP quick/JJ)
  and/CC
  (NP he/PRP)
  may/MD
  jump/VB
  over/IN
  (NP the/DT lazy/JJ dog/NN))
(S
  (NP He/PRP)
  (VP reckons/VBZ)
  (NP the/DT current/JJ account/NN deficit/NN)
  (VP will/MD narrow/VB)
  (PP to/TO)
  (NP only/RB #/# 1.8/CD billion/CD)
  (PP in/IN)
  (NP September/NNP)
  ./.)
C:\Users\User\AppData\Local\Temp\ipykernel_11252\1847143938.py:141: DeprecationWarnin
g:
  Function evaluate() has been deprecated.  Use accuracy(gold)
  instead.
  print(tc.evaluate(test_wsj_data))
```

```
ChunkParse score:
    IOB Accuracy:  89.4%%
    Precision:     80.8%%
    Recall:        86.0%%
    F-Measure:     83.3%%
```

In [5]:

```
# dependency_parsing.py
# os.environ['PATH'] = 'C:/Program Files/Graphviz/bin'
sentence = 'Colorless green ideas sleep furiously'

from spacy.lang.en import English
parser = English()
parsed_sent = parser(sentence)

dependency_pattern = '{left}<---{word}[{w_type}]--->{right}\n--------'
for token in parsed_sent:
    print(dependency_pattern.format(word=token.orth_,
                                    w_type=token.dep_,
                                    left=[t.orth_
                                              for t
                                              in token.lefts],
                                    right=[t.orth_
                                               for t
                                               in token.rights]))


# Since the Stanford Dependency parser is written in Java, we need to:
# 0. Install Java 1.8 or better
# 1. Download the Stanford Parser from https://nlp.stanford.edu/software/lex-parser.sh
# 2. Unzip it to a directory and
# 3. Set environment variables to our JAVA path and this direcory
# import os
# java_path = r'/usr/bin/java'
# os.environ['JAVAHOME'] = java_path

from nltk.parse.stanford import StanfordDependencyParser

"""
Previous version of NLTK/Stanford Parser syntax
sdp = StanfordDependencyParser(path_to_jar='/Users/arw/Documents/Work/stanford-parser-
                               model_path='/Users/arw/Documents/Work/stanford-parser-f
"""

sdp = StanfordDependencyParser(model_path='C:/Program Files/stanford-parser-full-2020-


result = list(sdp.raw_parse(sentence))

"""
result[0]

[item for item in result[0]]
"""

dep_tree = [parse for parse in result][0]
print(dep_tree)
dep_tree

# generation of annotated dependency tree
from graphviz import Source
```

```
dep_tree_dot_repr = [parse for parse in result][0].to_dot()
source = Source(dep_tree_dot_repr, filename="dep_tree", format="png")
source.view()

import nltk
tokens = nltk.word_tokenize(sentence)

dependency_rules = """
'fox' -> 'The' | 'brown'
'quick' -> 'fox' | 'is' | 'and' | 'jumping'
'jumping' -> 'he' | 'is' | 'dog'
'dog' -> 'over' | 'the' | 'lazy'
"""

dependency_grammar = nltk.grammar.DependencyGrammar.fromstring(dependency_rules)
print(dependency_grammar)

dp = nltk.ProjectiveDependencyParser(dependency_grammar)
res = [item for item in dp.parse(tokens)]
if len(res) > 0:
    tree = res[0]
    print(tree)
    tree.draw()
else:
    print("No valid parse trees were found.")
```

```
[]<---Colorless[]--->[]
--------
[]<---green[]--->[]
--------
[]<---ideas[]--->[]
--------
[]<---sleep[]--->[]
--------
[]<---furiously[]--->[]
--------
```

```
C:\Users\User\AppData\Local\Temp\ipykernel_11252\4249957290.py:38: DeprecationWarnin
g: The StanfordDependencyParser will be deprecated
Please use nltk.parse.corenlp.CoreNLPDependencyParser instead.
  sdp = StanfordDependencyParser(model_path='C:/Program Files/stanford-parser-full-20
20-11-17/edu/stanford/nlp/models/lexparser/englishPCFG.ser.gz')
```

```
defaultdict(<function DependencyGraph.__init__.<locals>.<lambda> at 0x000001580F0868C
0>,
            {0: {'address': 0,
                 'ctag': 'TOP',
                 'deps': defaultdict(<class 'list'>, {'root': [4]}),
                 'feats': None,
                 'head': None,
                 'lemma': None,
                 'rel': None,
                 'tag': 'TOP',
                 'word': None},
             1: {'address': 1,
                 'ctag': 'JJ',
                 'deps': defaultdict(<class 'list'>, {}),
                 'feats': '_',
                 'head': 3,
                 'lemma': '_',
                 'rel': 'amod',
                 'tag': 'JJ',
                 'word': 'Colorless'},
             2: {'address': 2,
                 'ctag': 'JJ',
                 'deps': defaultdict(<class 'list'>, {}),
                 'feats': '_',
                 'head': 3,
                 'lemma': '_',
                 'rel': 'amod',
                 'tag': 'JJ',
                 'word': 'green'},
             3: {'address': 3,
                 'ctag': 'NNS',
                 'deps': defaultdict(<class 'list'>, {'amod': [1, 2]}),
                 'feats': '_',
                 'head': 4,
                 'lemma': '_',
                 'rel': 'nsubj',
                 'tag': 'NNS',
                 'word': 'ideas'},
             4: {'address': 4,
                 'ctag': 'VBP',
                 'deps': defaultdict(<class 'list'>,
                                     {'advmod': [5],
                                      'nsubj': [3]}),
                 'feats': '_',
                 'head': 0,
                 'lemma': '_',
                 'rel': 'root',
                 'tag': 'VBP',
                 'word': 'sleep'},
             5: {'address': 5,
                 'ctag': 'RB',
                 'deps': defaultdict(<class 'list'>, {}),
                 'feats': '_',
                 'head': 4,
                 'lemma': '_',
                 'rel': 'advmod',
                 'tag': 'RB',
                 'word': 'furiously'}})
Dependency grammar with 12 productions
  'fox' -> 'The'
```

```
  'fox' -> 'brown'
  'quick' -> 'fox'
  'quick' -> 'is'
  'quick' -> 'and'
  'quick' -> 'jumping'
  'jumping' -> 'he'
  'jumping' -> 'is'
  'jumping' -> 'dog'
  'dog' -> 'over'
  'dog' -> 'the'
  'dog' -> 'lazy'
No valid parse trees were found.
```

In [7]:

```python
# shallow_parsing_with_pattern.py

from pattern.en import parsetree, Chunk
from nltk.tree import Tree

sentence = 'I saw the man with the telescope'

tree = parsetree(sentence)
print(tree)

for sentence_tree in tree:
    print(sentence_tree.chunks)

for sentence_tree in tree:
    for chunk in sentence_tree.chunks:
        print(chunk.type, '->', [(word.string, word.type)
                                 for word in chunk.words])


def create_sentence_tree(sentence, lemmatize=False):
    sentence_tree = parsetree(sentence,
                              relations=True,
                              lemmata=lemmatize)
    return sentence_tree[0]

def get_sentence_tree_constituents(sentence_tree):
    return sentence_tree.constituents()

def process_sentence_tree(sentence_tree):

    tree_constituents = get_sentence_tree_constituents(sentence_tree)
    processed_tree = [
                        (item.type,
                         [
                             (w.string, w.type)
                             for w in item.words
                         ]
                        )
                        if type(item) == Chunk
                        else ('-',
                              [
                                   (item.string, item.type)
                              ]
                             )
                             for item in tree_constituents
                    ]
```

```
    return processed_tree

def print_sentence_tree(sentence_tree):


    processed_tree = process_sentence_tree(sentence_tree)
    processed_tree = [
                        Tree( item[0],
                             [
                                 Tree(x[1], [x[0]])
                                 for x in item[1]
                             ]
                            )
                            for item in processed_tree
                    ]

    tree = Tree('S', processed_tree )
    print(tree)

def visualize_sentence_tree(sentence_tree):

    processed_tree = process_sentence_tree(sentence_tree)
    processed_tree = [
                        Tree( item[0],
                             [
                                 Tree(x[1], [x[0]])
                                 for x in item[1]
                             ]
                            )
                            for item in processed_tree
                    ]
    tree = Tree('S', processed_tree )
    tree.draw()


t = create_sentence_tree(sentence)
print(t)

pt = process_sentence_tree(t)
pt

print_sentence_tree(t)
visualize_sentence_tree(t)
```

```
I saw the man with the telescope
[Chunk('I/NP'), Chunk('saw/VP'), Chunk('the man/NP'), Chunk('with/PP'), Chunk('the te
lescope/NP')]
NP -> [('I', 'PRP')]
VP -> [('saw', 'VBD')]
NP -> [('the', 'DT'), ('man', 'NN')]
PP -> [('with', 'IN')]
NP -> [('the', 'DT'), ('telescope', 'NN')]
I saw the man with the telescope
(S
  (NP (PRP I))
  (VP (VBD saw))
  (NP (DT the) (NN man))
  (PP (IN with))
  (NP (DT the) (NN telescope)))
```

In [ ]: