Using NLTK w/CYK

Following assumes no ambiguity!

tokens = ['the', 'kids', 'opened', 'the', 'box', 'on', 'the', 'floor']

grammar = nltk.parse_cfg('"
S -> NP VP
PP -> P NP
NP -> Det N | NP PP
VP -> V NP | VP PP
Det -> 'the'
N -> 'kids' | 'box' | 'floor'
V -> 'opened'
P -> 'on'
"")

Initialize Table

def init_wfst(tokens, grammar):
    numtokens = len(tokens)
    for i in range(numtokens):
        productions = grammar.productions(rhs=tokens[i])
        wfst[i][i+1] = productions[0].lhs()
    return wfst

Fill In Table

def complete_wfst(wfst, tokens, trace=False):
    index = {}
    for prod in grammar.productions():
        index[prod.rhs()] = prod.lhs()
    for span in range(2, numtokens+1):
        for start in range(numtokens+1-span):
            end = start + span
            for mid in range(start+1, end):
                nt1, nt2 = wfst[start][mid], wfst[mid][end]
                if (nt1,nt2) in index:
                    if trace:
                        print '[%d] %3s [%d] %3s [%d] %3s' % 
                            (start, nt1, mid, nt2, end, start, index[(nt1,nt2)], end)
                    wfst[start][end] = index[(nt1,nt2)]
    return wfst

Display Table

def display(wfst, tokens):
    print 'WFST ' + ' '.join(['%4d' % i for i in range(1, len(wfst))])
    for i in range(len(wfst)-1):
        print '%d ' % i,
        for j in range(1, len(wfst)):
            print '%4s' % wfst[i][j],
        print
Results
tokens = ["the", "kids", "opened", "the", "box", "on", "the", "floor"]

```python
gdf = init_wfst(tokens, grammar)
display(gdf, tokens)
```

WFST 1 2 3 4 5 6 7 8
0  Det . . . . . . .
1  . N . . . . . .
2  . V . . . . . .
3  . Det . . . .
4  . . N . . . .
5  . . . P . . .
6  . . . Det . .
7  . . . . . . N

Results
tokens = ["the", "kids", "opened", "the", "box", "on", "the", "floor"]

```python
gdf = complete_wfst(gdf, tokens)
display(gdf, tokens)
```

WFST 1 2 3 4 5 6 7 8
0  Det NP . . S . S
1  . N . . . . . .
2  . V . VP . VP
3  . Det NP . NP
4  . . . N . . .
5  . . . . P . PP
6  . . . . Det NP
7  . . . . . . N

With Tracing
tokens = ["the", "kids", "opened", "the", "box", "on", "the", "floor"]

```python
gdf = complete_wfst(gdf, tokens, trace=True)
display(gdf, tokens)
```

Earley Algorithm

- Top-down
- Does not require CNF, handles left-recursion.
- Proceeds left-to-right filling in a chart
- States contain 3 pieces of info:
  - Grammar rule
  - Progress made in recognizing it
  - Position of subtree in input string

Parse Table

- As before, columns correspond to gaps
- Entry in column n of the form
  - A → u,v,k
    - Means predicting that we’ll use rule A → u v, and so far have verified u in input matches section of input [k,n]
- Ex: 0 Book 1 that 2 flight 3
  - NP → Det.Nom, 1 in column 2 means have recognized “that” (word[1,2]) is Det and hope to show Nom occurs later
**Earley Algorithm**

Add \( \text{ROOT} \rightarrow \cdot \text{S} \) to column 0.

For each \( j \) from 0 to \( n \):

For each dotted rule in column \( j \),

(including those added as we go!)

look at what’s after the dot:

- If it’s a word \( w \), SCAN:
  - If \( w \) matches the input word between \( j \) and \( j+1 \),
    advance the dot and add the new rule to column \( j+1 \)

- If it’s a non-terminal \( X \), PREDICT:
  - Add all rules for \( X \) to the bottom of column \( j \), with
    the dot at the start: e.g. \( X \rightarrow \cdot \text{YZ} \)

- If there’s nothing after the dot, ATTACH:
  - We’ve finished some constituent, \( A \), that started in
    column \( i<j \). So for each rule in column \( j \) that has \( A 
    
Return true if last column has \( \text{ROOT} \rightarrow \cdot \text{S} \).

---

**Idea of Algorithm**

- Process all hypotheses in order
- May add new hypotheses (or try to add old)
- Process according to what after dot
  - if word, scan and see if matches
  - if non-terminal, predict ways to match
  - if want, can be smart and peek ahead to reduce possibilities
- if at end, have complete constituent and attach to those that need it.

---

**Example**

<table>
<thead>
<tr>
<th>( S \rightarrow \cdot \text{NP VP} )</th>
<th>( \text{NP} \rightarrow \cdot \text{Det Nom} )</th>
<th>( \text{VP} \rightarrow \cdot \text{Verb} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow \cdot \text{Aux NP VP} )</td>
<td>( \text{NP} \rightarrow \cdot \text{Det Nom} )</td>
<td>( \text{VP} \rightarrow \cdot \text{Verb} )</td>
</tr>
<tr>
<td>( S \rightarrow \cdot \text{VP} )</td>
<td>( \text{NP} \rightarrow \cdot \text{Det Nom} )</td>
<td>( \text{VP} \rightarrow \cdot \text{Verb} )</td>
</tr>
<tr>
<td>( \text{NP} \rightarrow \cdot \text{Det Nom} )</td>
<td>( \text{VP} \rightarrow \cdot \text{Verb} )</td>
<td>( \text{NP} \rightarrow \cdot \text{Det Nom} )</td>
</tr>
<tr>
<td>( \text{VP} \rightarrow \cdot \text{Verb} )</td>
<td>( \text{NP} \rightarrow \cdot \text{Det Nom} )</td>
<td>( \text{VP} \rightarrow \cdot \text{Verb} )</td>
</tr>
</tbody>
</table>

*Book that flight!*

---

**Earley Example**

<table>
<thead>
<tr>
<th>chart[0]</th>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ROOT} \rightarrow \cdot \text{S, 0} )</td>
<td>( \text{book} )</td>
</tr>
<tr>
<td>( \text{S} \rightarrow \cdot \text{NP VP, 0} )</td>
<td>( \text{S} \rightarrow \cdot \text{Aux NP VP, 0} )</td>
</tr>
<tr>
<td>( \text{S} \rightarrow \cdot \text{VP} )</td>
<td>( \text{NP} \rightarrow \cdot \text{Det Nom, 0} )</td>
</tr>
<tr>
<td>( \text{NP} \rightarrow \cdot \text{Det Nom, 0} )</td>
<td>( \text{VP} \rightarrow \cdot \text{Verb, 0} )</td>
</tr>
<tr>
<td>( \text{VP} \rightarrow \cdot \text{Verb NP, 0} )</td>
<td>( \text{VP} \rightarrow \cdot \text{Verb NP, 0} )</td>
</tr>
</tbody>
</table>

*3 predictions for \( S \) because \( \text{book} \) is not Aux or Det*
Using NLTK

import nltk

grammar = nltk.parse_cfg('''
NP -> NNS | JJ NNS | NP CC NP
NNS -> "men" | "women" | "children" | NNS CC NNS
JJ -> "old" | "young"
CC -> "and" | "or"
''')

parser = nltk.ChartParser(grammar, nltk.parse.BU_STRATEGY)

Also TD_STRATEGY

Using NLTK to Parse

>>> sent = 'old men and women'.split()
>>> for tree in parser.nbest_parse(sent):
...    print tree
...
(NP (JJ old) (NNS (NNS men) (CC and) (NNS women))))
(NP (NP (JJ old) (NNS men)) (CC and) (NP (NNS women))))

Earley Example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>book</td>
<td>hat</td>
<td>flight</td>
<td></td>
</tr>
</tbody>
</table>

ROOT -> S . 0
Verb -> book . 0
Det -> that . 1
Noun -> flight . 2
S -> NP VP . 0
VP -> Verb . 0
Det -> Det NOM . 1
NOM -> Noun . 2
NOM -> Noun NOM . 2
S -> NP . 0
NP -> Det NOM . 0
NOM -> Noun NOM . 1
NP -> Det NOM NOM . 1
VP -> Verb . 0
S -> VP . 0
NOM -> Nom . 2
NOM -> Nom . 2

Complexity

- Size of table is n^nG
- Processing one cell might require search previous chart and check for dups.
- Total O(G^2n^3)

Using NLTK

import nltk

grammar = nltk.parse_cfg('''
NP -> NNS | JJ NNS | NP CC NP
NNS -> "men" | "women" | "children" | NNS CC NNS
JJ -> "old" | "young"
CC -> "and" | "or"
''')

parser = nltk.ChartParser(grammar, nltk.parse.BU_STRATEGY)

Also TD_STRATEGY

Statistical Parsing
**Why Use Probabilities In Parsing?**

- Disambiguation
- Language modeling -- fix errors
- Use probabilistic CFGs

**Assign probabilities to rules**
- Represent probability of using rule, given already have LHS.
- Rules from given LHS must add up to 1
  - \( VP \rightarrow \text{Verb} \quad 0.55 \)
  - \( VP \rightarrow \text{Verb NP} \quad 0.40 \)
  - \( VP \rightarrow \text{Verb NP NP} \quad 0.05 \)

**Computing Probabilities**
- Compute probability of tree by multiplying probabilities of rules used.
- Probability of sentence if sum of probabilities of all of its parse trees.
- Can read sentence off of parse tree.
- Sum of probabilities of all grammatical sentences should add up to 1 to have a consistent grammar
  - Problem: \( S \rightarrow SS, S \rightarrow a \)

**Disambiguating Sentences**
- Choose the parse tree with highest probability to disambiguate sentence.
- Just a first approximation!

**Probabilistic CYK**

```plaintext```
function PCKY_Parse(words, grammar)
  n ← length(words)
  for w ← 1 to n do
    table[w-1,w] ← \{ A \mid A \rightarrow \text{words}[w] \in grammar \}
  for start ← 0 to n-w do  # start is row
    end ← start + w  # end is column
    for mid ← start+1 to end-1
      for every X in table[start,mid]
        for every Y in table[mid,end]
          for all B s.t B \rightarrow X Y \in grammar
            add B, prob to table[start,end]
```
**Probabilistic Parses**

- Notice only need to keep at each node, parse tree for B w/max. probability if only want most likely parse.
- If want probability of all, then have to add each of them to table and keep track of probabilities.

**Getting Probabilities of Rules**

- If possible, use an annotated database (treebank)
- Penn Treebank has ~ 1.6 million words
- Available in other languages as well
- Collect count for each rule expansion and normalize:
  \[
P(\alpha \rightarrow \beta | \alpha) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\sum_{\gamma} \text{Count}(\alpha \rightarrow \gamma)}
\]

**Learning Probabilities**

- What if don’t have one for kind of corpus?
- Take large collection of text and parse.
- If ambiguous, keep all possible parses
- Guess relative probabilities for ambiguous How???
- Continue as with treebank

**Learn by approximations**

- Need probabilistic parser to assign probabilities to ambiguous parses!
- Most sentences are ambiguous!!
- One technique:
  - Start w/ all same probability
  - Compute new probability for each parse
  - repeat ...

**PCFG’s in NLTK**
import nltk

grammar = nltk.parse_pcfg('''
NP -> NNS [0.5] | JJ NNS [0.3] | NP CC NP [0.2]
NNS -> "men" [0.1] | "women" [0.2] | "children" [0.3] | NNS CC NNS [0.4]
JJ -> "old" [0.4] | "young" [0.6]
CC -> "and" [0.9] | "or" [0.1]
''')

viterbi_parser = nltk.ViterbiParser(grammar)

>>> sent = 'old men and women'.split()
>>> print viterbi_parser.parse(sent)
(NP (JJ old) (NNS (NNS men) (CC and) (NNS women))) (p=0.000864)