Today

- Parsing with CFGs
  - Bottom-up, top-down
  - Ambiguity
  - CKY parsing
  - Earley
  - Shallow
Parsing

- Parsing with CFGs refers to the task of assigning proper trees to input strings
- Proper here means a tree that covers all and only the elements of the input and has an S at the top
- It doesn’t actually mean that the system can select the correct tree from among all the possible trees

Parsing

- As with everything of interest, parsing involves a search which involves the making of choices
- We’ll start with some basic (meaning bad) methods before moving on to the one or two that you need to know
For Now

- Assume...
  - You have all the words already in some buffer
  - The input isn’t POS tagged
  - We won’t worry about morphological analysis
  - All the words are known

- These are all problematic in various ways, and would have to be addressed in real applications.

Top-Down Search

- Since we’re trying to find trees rooted with an $S$ (Sentences), why not start with the rules that give us an $S$.
- Then we can work our way down from there to the words.
Bottom-Up Parsing

- Of course, we also want trees that cover the input words. So we might also start with trees that link up with the words in the right way.
- Then work your way up from there to larger and larger trees.
Bottom-Up Search

Book that flight

Verb  Det  Noun

Book  that  flight
Bottom-Up Search

Nominal

Verb  Det  Noun

Book  that  flight

Bottom-Up Search

NP

Nominal

Verb  Det  Noun

Book  that  flight
Bottom-Up Search

```
VP
  NP
  Verb Det Noun
Book that flight
```

"The old dog the footsteps of the young."

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td>VP → V</td>
</tr>
<tr>
<td>S → Aux NP VP</td>
<td>VP → V PP</td>
</tr>
<tr>
<td>S → VP</td>
<td>PP → Prep NP</td>
</tr>
<tr>
<td>NP → Det Nom</td>
<td>N → old</td>
</tr>
<tr>
<td>NP → PropN</td>
<td>V → dog</td>
</tr>
<tr>
<td>Nom → Adj N</td>
<td>Aux → does</td>
</tr>
<tr>
<td>Nom → N</td>
<td>Prep → from</td>
</tr>
<tr>
<td>Nom → N Nom</td>
<td>PropN → Fido</td>
</tr>
<tr>
<td>Nom → Nom PP</td>
<td>Det → that</td>
</tr>
<tr>
<td>VP → V NP</td>
<td>Adj → old</td>
</tr>
</tbody>
</table>
Top-Down and Bottom-Up

- **Top-down**
  - Only searches for trees that can be answers (i.e. S’s)
  - But also suggests trees that are not consistent with any of the words

- **Bottom-up**
  - Only forms trees consistent with the words
  - But suggests trees that make no sense globally

Control

- Of course, in both cases we left out how to keep track of the search space and how to make choices
  - Which node to try to expand next
  - Which grammar rule to use to expand a node

- One approach is called backtracking.
  - Make a choice, if it works out then fine
  - If not then back up and make a different choice
Problems

- Even with the best filtering, backtracking methods are doomed because of two inter-related problems
  - Ambiguity
  - Shared subproblems

Ambiguity
Example types of ambiguity

- POS
- Attachment
  - PP
  - Coordination (*old dogs and cats*)

Shared Sub-Problems

- No matter what kind of search (top-down or bottom-up or mixed) that we choose.
  - We don’t want to redo work we’ve already done.
  - Unfortunately, naïve backtracking will lead to duplicated work.
Shared Sub-Problems

- Consider
  - A flight from Indianapolis to Houston on TWA

```
NP
  /   \
/     \  
Det a   Nominal
          /   \  
          /     \  
          Nominal PP
              /   \  
              /     \  
              Nominal PP
                 /   \  
                 /     \  
                 Nominal PP
                    /   \  
                    /     \  
                    Nominal PP
```

- Assume a top-down parse making choices among the various Nominal rules.
- In particular, between these two
  - Nominal -> Noun
  - Nominal -> Nominal PP
- Statically choosing the rules in this order leads to the following bad results...
Shared Sub-Problems

NP
  |   |
Det   Nominal
  |   |
a   Noun
     |
  flight...
Shared Sub-Problems

1. NP
2. Det
3. "a"
4. Nominal
5. Nominal
6. Nominal
7. Noun
8. "flight"
9. PP
10. "to Houston..."

Shared Sub-Problems

1. NP
2. Det
3. "a"
4. Nominal
5. Nominal
6. Nominal
7. Noun
8. "from Indianapolis"
9. PP
10. "to Houston"
11. PP
12. "on TWA"

10/7/2013 Speech and Language Processing - Jurafsky and Martin
Dynamic Programming

- DP search methods fill tables with partial results and thereby
  - Avoid doing avoidable repeated work
  - Solve exponential problems in polynomial time
  - Efficiently store ambiguous structures with shared sub-parts.

- We’ll cover two approaches that roughly correspond to bottom-up and top-down approaches.
  - CKY
  - Earley

CKY Parsing

- First we’ll limit our grammar to epsilon-free, binary rules (more later)
- Consider the rule \( A \rightarrow BC \)
  - If there is an A somewhere in the input then there must be a B followed by a C in the input.
  - If the A spans from i to j in the input then there must be some k st. \( i < k < j \)
    - Ie. The B splits from the C someplace.
Problem

- What if your grammar isn’t binary?
  - As in the case of the TreeBank grammar?
- Convert it to binary... any arbitrary CFG can be rewritten into Chomsky-Normal Form automatically.
- What does this mean?
  - The resulting grammar accepts (and rejects) the same set of strings as the original grammar.
  - But the resulting derivations (trees) are different.

Problem

- More specifically, we want our rules to be of the form
  \[ A \rightarrow B \ C \]
  Or
  \[ A \rightarrow w \]

*That is, rules can expand to either 2 non-terminals or to a single terminal.*
Binarization Intuition

- Eliminate chains of unit productions.
- Introduce new intermediate non-terminals into the grammar that distribute rules with length > 2 over several rules.
  - So... $S \rightarrow A B C$ turns into
    $S \rightarrow X C$ and
    $X \rightarrow A B$
  Where $X$ is a symbol that doesn’t occur anywhere else in the the grammar.

Sample L1 Grammar

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Lexicon</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow$ NP VP</td>
<td>Det $\rightarrow$ that</td>
</tr>
<tr>
<td>$S \rightarrow$ Aux NP VP</td>
<td>Noun $\rightarrow$ book</td>
</tr>
<tr>
<td>$S \rightarrow$ VP</td>
<td>Verb $\rightarrow$ book</td>
</tr>
<tr>
<td>NP $\rightarrow$ Pronoun</td>
<td>Pronoun $\rightarrow$ I</td>
</tr>
<tr>
<td>NP $\rightarrow$ Proper-Noun</td>
<td>Proper-Noun $\rightarrow$ Houston</td>
</tr>
<tr>
<td>NP $\rightarrow$ Det Nominal</td>
<td>Aux $\rightarrow$ does</td>
</tr>
<tr>
<td>Nominal $\rightarrow$ Noun</td>
<td>Preposition $\rightarrow$ from</td>
</tr>
<tr>
<td>Nominal $\rightarrow$ Nominal Noun</td>
<td></td>
</tr>
<tr>
<td>Nominal $\rightarrow$ Nominal PP</td>
<td></td>
</tr>
<tr>
<td>VP $\rightarrow$ Verb</td>
<td></td>
</tr>
<tr>
<td>VP $\rightarrow$ Verb NP</td>
<td></td>
</tr>
<tr>
<td>VP $\rightarrow$ Verb NP PP</td>
<td></td>
</tr>
<tr>
<td>VP $\rightarrow$ Verb PP</td>
<td></td>
</tr>
<tr>
<td>VP $\rightarrow$ VP PP</td>
<td></td>
</tr>
<tr>
<td>PP $\rightarrow$ Preposition NP</td>
<td></td>
</tr>
</tbody>
</table>
**CNF Conversion**

<table>
<thead>
<tr>
<th>$\mathcal{L}_G$ Grammar</th>
<th>$\mathcal{L}_A$ in CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow NP \ VP$</td>
<td>$S \rightarrow NP \ VP$</td>
</tr>
<tr>
<td>$S \rightarrow Aux \ NP \ VP$</td>
<td>$S \rightarrow X1 \ VP$</td>
</tr>
<tr>
<td>$S \rightarrow VP$</td>
<td>$XI \rightarrow Aux \ NP$</td>
</tr>
<tr>
<td>$S \rightarrow book \mid include \mid prefer$</td>
<td>$S \rightarrow Verb \ NP$</td>
</tr>
<tr>
<td>$S \rightarrow X2 \ PP$</td>
<td>$S \rightarrow Verb \ PP$</td>
</tr>
<tr>
<td>$S \rightarrow VP \ PP$</td>
<td>$S \rightarrow VP \ PP$</td>
</tr>
<tr>
<td>$NP \rightarrow Pronoun$</td>
<td>$NP \rightarrow I \mid she \mid me$</td>
</tr>
<tr>
<td>$NP \rightarrow Proper \ Noun$</td>
<td>$NP \rightarrow TWA \mid Houston$</td>
</tr>
<tr>
<td>$NP \rightarrow Det \ Nominal$</td>
<td>$NP \rightarrow Det \ Nominal$</td>
</tr>
<tr>
<td>$Nominal \rightarrow Noun$</td>
<td>$Nominal \rightarrow book \mid flight \mid meal \mid money$</td>
</tr>
<tr>
<td>$Nominal \rightarrow Nominal \ Noun$</td>
<td>$Nominal \rightarrow Nominal \ Noun$</td>
</tr>
<tr>
<td>$Nominal \rightarrow Nominal \ PP$</td>
<td>$Nominal \rightarrow Nominal \ PP$</td>
</tr>
<tr>
<td>$VP \rightarrow Verb$</td>
<td>$VP \rightarrow book \mid include \mid prefer$</td>
</tr>
<tr>
<td>$VP \rightarrow Verb \ NP$</td>
<td>$VP \rightarrow Verb \ NP$</td>
</tr>
<tr>
<td>$VP \rightarrow Verb \ NP \ PP$</td>
<td>$VP \rightarrow X2 \ PP$</td>
</tr>
<tr>
<td>$VP \rightarrow Verb \ PP$</td>
<td>$X2 \rightarrow Verb \ NP$</td>
</tr>
<tr>
<td>$VP \rightarrow VP \ PP$</td>
<td>$VP \rightarrow VP \ PP$</td>
</tr>
<tr>
<td>$PP \rightarrow Preposition \ NP$</td>
<td>$PP \rightarrow Preposition \ NP$</td>
</tr>
</tbody>
</table>

---

**CKY**

- So let’s build a table so that an A spanning from $i$ to $j$ in the input is placed in cell $[i,j]$ in the table.
- So a non-terminal spanning an entire string will sit in cell $[0, n]$
  - Hopefully an $S$
- If we build the table bottom-up, we’ll know that the parts of the A must go from $i$ to $k$ and from $k$ to $j$, for some $k$. 
CKY

- Meaning that for a rule like $A \rightarrow B \ C$ we should look for a $B$ in $[i,k]$ and a $C$ in $[k,j]$.
- In other words, if we think there might be an $A$ spanning $i,j$ in the input... AND $A \rightarrow B \ C$ is a rule in the grammar THEN
- There must be a $B$ in $[i,k]$ and a $C$ in $[k,j]$ for some $i<k<j$

---

CKY

- So to fill the table loop over the cell$[i,j]$ values in some systematic way
  - What constraint should we put on that systematic search?

  - For each cell, loop over the appropriate $k$ values to search for things to add.
Note

- We arranged the loops to fill the table a column at a time, from left to right, bottom to top.
  - This assures us that whenever we’re filling a cell, the parts needed to fill it are already in the table (to the left and below)
  - It’s somewhat natural in that it processes the input a left to right a word at a time
    - Known as online

Example
CKY Parser

Cell $[i,j]$ contains all constituents (non-terminals) covering words $i+1$ through $j$
CKY Parser

Book the flight through Houston

S, VP, Verb, Nominal, Noun

VP

None

Det

Nominal, Noun

NP

CKY Parser

Book the flight through Houston

S, VP, Verb, Nominal, Noun

VP

None

Det

Nominal, Noun

NP
### CKY Parser

#### Book the flight through Houston

<table>
<thead>
<tr>
<th></th>
<th>S, VP, Verb, Nominal, Noun</th>
<th>None</th>
<th>S VP, X2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Det</td>
<td>Nominal, Noun</td>
<td>NP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>None</td>
</tr>
</tbody>
</table>
CKY Parser

Book the flight through Houston

<table>
<thead>
<tr>
<th>S, VP, Verb, Nominal, Noun</th>
<th>S VP</th>
<th>None</th>
<th>S VP</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Det</td>
<td>NP</td>
<td>None</td>
<td>NP</td>
<td>Nominal</td>
</tr>
<tr>
<td>Nominal, Noun</td>
<td>None</td>
<td>Nominal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prep</td>
<td>PP</td>
<td>NP</td>
<td>ProperNoun</td>
<td></td>
</tr>
</tbody>
</table>

CKY Parser

Book the flight through Houston

<table>
<thead>
<tr>
<th>S, VP, Verb, Nominal, Noun</th>
<th>S VP</th>
<th>None</th>
<th>S VP</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Det</td>
<td>NP</td>
<td>None</td>
<td>NP</td>
<td>Nominal</td>
</tr>
<tr>
<td>Nominal, Noun</td>
<td>None</td>
<td>Nominal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prep</td>
<td>PP</td>
<td>NP</td>
<td>ProperNoun</td>
<td></td>
</tr>
</tbody>
</table>
CKY Parser

Book       the        flight    through Houston

S, VP, Verb,
Nominal,
Noun

None

S

None

S

VP

Det

NP

None

VP

Nominal,
Noun

None

Nominal

Prep

PP

NP

ProperNoun

CKY Parser

Book       the        flight    through Houston

S, VP, Verb,
Nominal,
Noun

None

S

None

S

VP

Det

NP

None

VP

Nominal,
Noun

None

Nominal

Prep

PP

NP

ProperNoun
Example

Example
CKY Notes

- Since it’s bottom up, CKY populates the table with a lot of phantom constituents.
  - Segments that by themselves are constituents but cannot really occur in the context in which they are being suggested.
  - To avoid this we can switch to a top-down control strategy
  - Or we can add some kind of filtering that blocks constituents where they can not happen in a final analysis.
Earley Parsing

- Allows arbitrary CFGs
- Top-down control
- Fills a table in a single sweep over the input
  - Table is length N+1; N is number of words
  - Table entries represent
    - Completed constituents and their locations
    - In-progress constituents
    - Predicted constituents

States

- The table-entries are called states and are represented with dotted-rules.
  
  \[
  \begin{align*}
  S & \rightarrow \cdot VP \quad \text{A VP is predicted} \\
  NP & \rightarrow \text{Det} \cdot \text{Nominal} \quad \text{An NP is in progress} \\
  VP & \rightarrow V \ NP \cdot \quad \text{A VP has been found}
  \end{align*}
  \]
States/Locations

- S $\rightarrow$ VP $[0,0]$
  - A VP is predicted at the start of the sentence

- NP $\rightarrow$ Det $\cdot$ Nominal $[1,2]$
  - An NP is in progress; the Det goes from 1 to 2

- VP $\rightarrow$ V NP $\cdot$ $[0,3]$
  - A VP has been found starting at 0 and ending at 3

Earley

- As with most dynamic programming approaches, the answer is found by looking in the table in the right place.
- In this case, there should be an S state in the final column that spans from 0 to N and is complete. That is,
  - S $\rightarrow$ $\alpha$ $\cdot$ $[0,N]$
- If that’s the case you’re done.
Earley

- So sweep through the table from 0 to N...
  - New predicted states are created by starting top-down from $S$
  - New incomplete states are created by advancing existing states as new constituents are discovered
  - New complete states are created in the same way.

Earley

- More specifically...
  1. Predict all the states you can upfront
  2. Read a word
     1. Extend states based on matches
     2. Generate new predictions
     3. Go to step 2
  3. When you’re out of words, look at the chart to see if you have a winner
Filling in the Chart

- March through chart left-to-right.
- At each step, apply 1 of 3 operators
  - **Predictor**
    - Create new states representing top-down expectations
  - **Scanner**
    - Match word predictions (rule with *POS* following dot) to words in input
  - **Completer**
    - When a state is complete, see what rules were looking for that complete constituent

Predictor

- Given a state
  - With a non-terminal to right of dot (not a part-of-speech category)
  - Create a new state for each expansion of the non-terminal
  - Put predicted states in same chart cell as generating state, beginning and ending where generating state ends
  - So predictor looking at
    - $S \rightarrow . \text{VP} [0,0]$
  - results in
    - $\text{VP} \rightarrow . \text{Verb} [0,0]$
    - $\text{VP} \rightarrow . \text{Verb \ NP} [0,0]$
Scanner

- Given a state
  - With a non-terminal to right of dot that is a POS category
  - If next word in input matches this POS
  - Create a new state with dot moved past the non-terminal
- E.g., scanner looking at VP -> . Verb NP [0,0]
  - If next word can be a verb, add new state:
    - VP -> Verb . NP [0,1]
  - Add this state to chart entry following current one
- NB: Earley uses top-down input to disambiguate POS -- only POS predicted by some state can be added to chart.

Completer

- Given a state
  - Whose dot has reached right end of rule
  - Parser has discovered a constituent over some span of input
  - Find and advance all previous states that are ‘looking for’ this category
  - Copy state, move dot, insert in current chart entry
- E.g., if processing:
  - NP -> Det Nominal . [1,3] and if state expecting an NP like VP -> Verb. NP [0,1] in chart
  - Add
    - VP -> Verb NP . [0,3] to same cell of chart
Reaching a Final State

- Find an S state in chart that spans input from 0 to N+1 and is complete
- Declare victory:
  - \( S \rightarrow \alpha \cdot [0,N+1] \)

Converting from Recognizer to Parser

- Augment the “Completer” to include pointer to each previous (now completed) state
- Read off all the backpointers from every complete S