Today

- Parsing with CFGs
  - Bottom-up, top-down
  - Ambiguity
  - CKY parsing
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
    Nominal
      Verb
      Det
        Book
      Noun
        that
        flight
```

**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
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

![Diagram showing a parse tree for a flight description.]

- 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

NP

Det

a

Nominal

Nominal

Nominal

PP

to Houston...

from Indianapolis

Noun

flight

Noun

f

Shared Sub-Problems

NP

Det

a

Nominal

Nominal

Nominal

PP

on TWA

to Houston

from Indianapolis

Noun

flight
Dynamic Programming

- DP search methods fill tables with partial results and thereby
  - Avoid doing avoidable repeated work
  - Solve exponential problems in polynomial time (well, no not really)
  - Efficiently store ambiguous structures with shared sub-parts.
- We’ll cover two approaches that roughly correspond to top-down and bottom-up 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 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 → that</td>
</tr>
<tr>
<td>$S \rightarrow Aux \ NP \ VP$</td>
<td>Noun → book</td>
</tr>
<tr>
<td>$S \rightarrow VP$</td>
<td>Verb → book</td>
</tr>
<tr>
<td>$NP \rightarrow Pronoun$</td>
<td>Pronoun → I</td>
</tr>
<tr>
<td>$NP \rightarrow Proper-Noun$</td>
<td>Proper-Noun → Houston</td>
</tr>
<tr>
<td>$NP \rightarrow Det \ Nominal$</td>
<td>Aux → does</td>
</tr>
<tr>
<td>Nominal → Noun</td>
<td>Preposition → from</td>
</tr>
<tr>
<td>Nominal → Nominal Noun</td>
<td></td>
</tr>
<tr>
<td>Nominal → Nominal PP</td>
<td></td>
</tr>
<tr>
<td>VP → Verb</td>
<td></td>
</tr>
<tr>
<td>VP → Verb NP</td>
<td></td>
</tr>
<tr>
<td>VP → Verb NP PP</td>
<td></td>
</tr>
<tr>
<td>VP → Verb PP</td>
<td></td>
</tr>
<tr>
<td>VP → VP PP</td>
<td></td>
</tr>
<tr>
<td>PP → Preposition NP</td>
<td></td>
</tr>
</tbody>
</table>
### CNF Conversion

<table>
<thead>
<tr>
<th>$Z_1$ Grammar</th>
<th>$Z_1$ 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>$S \rightarrow XI \ VP$</td>
</tr>
<tr>
<td>$S \rightarrow VP$</td>
<td>$XI \rightarrow Aux \ NP$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow book \</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow Verb \ NP$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow X2 \ PP$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow Verb \ PP$</td>
</tr>
<tr>
<td></td>
<td>$S \rightarrow VP \ PP$</td>
</tr>
<tr>
<td>$NP \rightarrow Pronoun$</td>
<td>$NP \rightarrow I \</td>
</tr>
<tr>
<td>$NP \rightarrow Proper-Noun$</td>
<td>$NP \rightarrow TWA \</td>
</tr>
<tr>
<td>$NP \rightarrow Det \ Nominal$</td>
<td>$NP \rightarrow Det \ Nominal$</td>
</tr>
<tr>
<td>$Nominal \rightarrow Noun$</td>
<td>$Nominal \rightarrow book \</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>$VP \rightarrow book \</td>
</tr>
<tr>
<td>$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.
**CKY Algorithm**

```
function CKY-PARSE(words, grammar) returns table

for j ← from 1 to LENGTH(words) do
    table[j - 1, j] ← \{A | A → words[j] ∈ grammar\}

for i ← from j - 2 downto 0 do
    for k ← i + 1 to j - 1 do
        \(table[i, j] ← table[i, j] \cup \{A | A → BC ∈ grammar, \)
        \(B ∈ table[i, k], \)
        \(C ∈ table[k, j]\}\)
```

**CKY Parsing**

- Is that really a parser?
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

<table>
<thead>
<tr>
<th>Book</th>
<th>the</th>
<th>flight</th>
<th>through</th>
<th>Houston</th>
</tr>
</thead>
<tbody>
<tr>
<td>S, VP, Verb, Nominal, Noun</td>
<td>S,VP,X2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Det</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nominal, Noun</td>
<td>Nominal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prep</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP, Proper-Noun</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Filling column 5
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.

  S $\rightarrow$ $\cdot$ VP $\quad$ A VP is predicted

  NP $\rightarrow$ Det $\cdot$ Nominal $\quad$ An NP is in progress

  VP $\rightarrow$ V NP $\cdot$ $\quad$ A VP has been found
States/Locations

- **S** $\rightarrow$ VP [0,0]
  - A VP is predicted at the start of the sentence
- **NP** $\rightarrow$ Det • Nominal [1,2]
  - An NP is in progress; the Det goes from 1 to 2
- **VP** $\rightarrow$ V NP • [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** $\rightarrow$ $\alpha$ • [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
Core Earley Code

function EARLEY-PARSE(words, grammar) returns chart

ENQUEUE(\(\gamma \rightarrow \bullet S, [0,0]), \text{chart}(0))
for \(i\) from 0 to LENGTH(words) do
  for each state in \text{chart}[i] do
    if INCOMPLETE?(state) and
      NEXT-CAT(state) is not a part of speech then
      PREDICTOR(state)
    elseif INCOMPLETE?(state) and
      NEXT-CAT(state) is a part of speech then
      SCANNER(state)
    else
      COMPLETER(state)
  end
end

return(chart)

Earley Code

procedure PREDICTOR((A \rightarrow \alpha \bullet B \beta, [i,j]))
  for each (B \rightarrow \gamma) in GRAMMAR-RULES-FOR(B, grammar) do
    ENQUEUE((B \rightarrow \bullet \gamma, [j,j]), \text{chart}[j])
  end

procedure SCANNER((A \rightarrow \alpha \bullet B \beta, [i,j]))
  if B \subseteq \text{PARTS-OF-SPEECH(word[j])} then
    ENQUEUE((B \rightarrow \text{word}[j], [j,j+1]), \text{chart}[j+1])
  end

procedure COMPLETER((B \rightarrow \gamma \bullet, [j,k]))
  for each (A \rightarrow \alpha \bullet B \beta, [i,j]) in \text{chart}[j] do
    ENQUEUE((A \rightarrow \alpha B \bullet \beta, [i,k]), \text{chart}[k])
  end
Example

- Book that flight
- We should find... an S from 0 to 3 that is a completed state...

Chart[0]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>$\gamma \rightarrow \bullet S$</td>
<td>[0,0]</td>
<td>Dummy start state</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>$S \rightarrow \bullet NP \ VP$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>$S \rightarrow \bullet \ Aux \ NP \ VP$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>$S \rightarrow \bullet \ VP$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>$NP \rightarrow \bullet Pronoun$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>$NP \rightarrow \bullet Proper-Noun$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>$NP \rightarrow \bullet Det \ Nominal$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>$VP \rightarrow \bullet \ Verb$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>$VP \rightarrow \bullet \ Verb \ NP$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td>$VP \rightarrow \bullet \ Verb \ NP \ PP$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S10</td>
<td>$VP \rightarrow \bullet \ VP \ PP$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S11</td>
<td>$VP \rightarrow \bullet \ VP \ PP$</td>
<td>[0,0]</td>
<td>Predictor</td>
<td></td>
</tr>
</tbody>
</table>

Note that given a grammar, these entries are the same for all inputs; they can be pre-loaded.
### Chart[1]

<table>
<thead>
<tr>
<th>Rule</th>
<th>A → B • C</th>
<th>Score</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>S12</td>
<td>Verb → book •</td>
<td>[0,1]</td>
<td>Scanner</td>
</tr>
<tr>
<td>S13</td>
<td>VP → Verb •</td>
<td>[0,1]</td>
<td>Completer</td>
</tr>
<tr>
<td>S14</td>
<td>VP → Verb • NP</td>
<td>[0,1]</td>
<td>Completer</td>
</tr>
<tr>
<td>S15</td>
<td>VP → Verb • NP PP</td>
<td>[0,1]</td>
<td>Completer</td>
</tr>
<tr>
<td>S16</td>
<td>VP → Verb • PP</td>
<td>[0,1]</td>
<td>Completer</td>
</tr>
<tr>
<td>S17</td>
<td>S → VP •</td>
<td>[0,1]</td>
<td>Completer</td>
</tr>
<tr>
<td>S18</td>
<td>VP → VP • PP</td>
<td>[0,1]</td>
<td>Completer</td>
</tr>
<tr>
<td>S19</td>
<td>NP → • Pronoun</td>
<td>[1,1]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S20</td>
<td>NP → • Proper-Noun</td>
<td>[1,1]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S21</td>
<td>NP → • Det Nominal</td>
<td>[1,1]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S22</td>
<td>PP → • Prep NP</td>
<td>[1,1]</td>
<td>Predictor</td>
</tr>
</tbody>
</table>

### Charts[2] and [3]

<table>
<thead>
<tr>
<th>Rule</th>
<th>A → B • C</th>
<th>Score</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>S23</td>
<td>Det → that •</td>
<td>[1,2]</td>
<td>Scanner</td>
</tr>
<tr>
<td>S24</td>
<td>NP → Det • Nominal</td>
<td>[1,2]</td>
<td>Completer</td>
</tr>
<tr>
<td>S25</td>
<td>Nominal → • Noun</td>
<td>[2,2]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S26</td>
<td>Nominal → • Nominal Noun</td>
<td>[2,2]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S27</td>
<td>Nominal → • Nominal PP</td>
<td>[2,2]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S28</td>
<td>Noun → fight •</td>
<td>[2,3]</td>
<td>Scanner</td>
</tr>
<tr>
<td>S29</td>
<td>Nominal → Noun •</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S30</td>
<td>NP → Det Nominal •</td>
<td>[1,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S31</td>
<td>Nominal → Nominal • Noun</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S32</td>
<td>Nominal → Nominal • PP</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S33</td>
<td>VP → Verb NP •</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S34</td>
<td>VP → Verb NP • PP</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S35</td>
<td>PP → • Prep NP</td>
<td>[3,3]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S36</td>
<td>S → VP •</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S37</td>
<td>VP → VP • PP</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
</tbody>
</table>
Efficiency

- For such a simple example, there seems to be a lot of useless stuff in there.
- Why?

  - It’s predicting things that aren’t consistent with the input
  - That’s the flipside to the CKY problem.

Details

- As with CKY that isn’t a parser until we add the backpointers so that each state knows where it came from.
Back to Ambiguity

- Did we solve it?

Ambiguity

- No...
  - Both CKY and Earley will result in multiple S structures for the [0,N] table entry.
  - They both efficiently store the sub-parts that are shared between multiple parses.
  - And they obviously avoid re-deriving those sub-parts.
  - But neither can tell us which one is right.
Ambiguity

- In most cases, humans don’t notice incidental ambiguity (lexical or syntactic). It is resolved on the fly and never noticed.
- We’ll try to model that with probabilities.
- But note something odd and important about the Groucho Marx example...