Two views of linguistic structure:

1. Constituency (phrase structure)
   - Phrase structure organizes words into nested constituents.
   - How do we know what is a constituent? (Not that linguists don’t argue about some cases.)
     - Distribution: a constituent behaves as a unit that can appear in different places:
       - John talked [to the children] about drugs.
       - *John talked (about drugs) to the children.
     - Distribution (expansion/pro-forms):
       - I sat [on the box/right on top of the box/there].

2. Dependency structure
   - Dependency structure shows which words depend on (modify or are arguments of) which other words.
     - The boy put the tortoise on the rug.
Two views of linguistic structure:  
2. Dependency structure  
• Dependency structure shows which words depend on (modify or are arguments of) which other words.

A Brief Parsing History  

Pre 1990 ("Classical") NLP Parsing  
• Wrote symbolic grammar (CFG or often richer) and lexicon  
  \[
  S \rightarrow NP\ VP \\
  NP \rightarrow (DT)\ NN \\
  NP \rightarrow NN\ NN \\
  VP \rightarrow V\ NP
  \]
• Used grammar/proof systems to prove parses from words  
• This scaled very badly and didn't give coverage. For sentence:  
  \[
  \text{Fed raises interest rates 0.5% in effort to control inflation}
  \]
  * Minimal grammar: 36 parses  
  * Simple 10 rule grammar: 592 parses  
  * Real-size broad-coverage grammar: millions of parses

Classical NLP Parsing:  
The problem and its solution  
• Categorical constraints can be added to grammars to limit unlikely/weird parses for sentences  
  * But the attempt make the grammars not robust  
  * In traditional systems, commonly 30% of sentences in even an edited text would have no parse.  
• A less constrained grammar can parse more sentences  
  * But simple sentences end up with ever more parses with no way to choose between them  
• We need mechanisms that allow us to find the most likely parse(s) for a sentence  
  * Statistical parsing lets us work with very loose grammars that admit millions of parses for sentences but still quickly find the best parse(s)

The rise of annotated data:  
The Penn Treebank  
[Marcus et al. 1993, Computational Linguistics]  

The rise of annotated data  
• Starting off, building a treebank seems a lot slower and less useful than building a grammar  
  * But a treebank gives us many things  
  * Reusability of the labor  
  * Many parsers, POS taggers, etc.  
  * Valuable resource for linguistics  
  * Broad coverage  
  * Frequencies and distributional information  
  * A way to evaluate systems
Statistical parsing applications

Statistical parsers are now robust and widely used in larger NLP applications:
• High precision question answering [Peca and Harabagiu SIGIR 2001]
• Improving biological named entity finding [Kiskel et al. JMLR 2004]
• Syntactically based sentence compression [Sun and Wilbur 2007]
• Extracting opinions about products [Bloom et al. NAACL 2007]
• Improved interaction in computer games [Jernite and Ray 2006]
• Helping linguists find data [Kiskel et al. MLJ 2005]
• Source sentence analysis for machine translation [Su et al. 2009]
• Relation extraction systems [Fundel et al. Bioinformatics 2006]

An exponential number of attachments

Attachment ambiguities

• A key parsing decision is how we ‘attach’ various constituents
  • PPs, adverbial or participial phrases, infinitives, coordinations, etc.

  The board approved [its acquisition] [by Royal Trustco Ltd.]
  [of Toronto]
  [for $27 a share]
  [at its monthly meeting].

• Catalan numbers: $C_n = \frac{(2n)!}{n! \cdot (n+1)!}$
• An exponentially growing series, which arises in many tree-like contexts:
  • E.g., the number of possible triangulations of a polygon with $n+2$ sides
  • Turns up in triangulation of probabilistic graphical models….

Two problems to solve:

1. Repeated work...

Two problems to solve:

2. Choosing the correct parse

• How do we work out the correct attachment:
  • She saw the man with a telescope
  • Is the problem ‘AI complete’? Yes, but …
  • Words are good predictors of attachment
    • E.g. absence full understanding
  • Moscow sent more than 100,000 soldiers into Afghanistan …
  • Sydney Water breached an agreement with NSW Health …
  • Our statistical parsers will try to exploit such statistics.
(Probabilistic) Context-Free Grammars

- CFG
- PCFG

Phrase structure grammars = context-free grammars (CFGs)

- G = (T, N, S, R)
  - T is a set of terminal symbols
  - N is a set of nonterminal symbols
  - S is the start symbol (S ∈ N)
  - R is a set of rules/productions of the form X → y
  - X ∈ N and y ∈ (N ∪ T)*
- A grammar G generates a language L.

A phrase structure grammar

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow V \ NP \\
NP & \rightarrow NP \ PP \\
NP & \rightarrow N \\
NP & \rightarrow e \\
PP & \rightarrow P \ NP \\
people \ fish \ tanks & \\
people \ fish \ with \ rods &
\end{align*}
\]

\[
\begin{align*}
N & \rightarrow \text{people} \\
N & \rightarrow \text{fish} \\
N & \rightarrow \text{tanks} \\
N & \rightarrow \text{rods} \\
V & \rightarrow \text{people} \\
V & \rightarrow \text{fish} \\
V & \rightarrow \text{tanks} \\
P & \rightarrow \text{with}
\end{align*}
\]

Probabilistic – or stochastic – context-free grammars (PCFGs)

- G = (T, N, S, R, P)
  - T is a set of terminal symbols
  - N is a set of nonterminal symbols
  - S is the start symbol (S ∈ N)
  - R is a set of rules/productions of the form X → y
  - P is a probability function
  - \( P : R \rightarrow [0, 1] \)
  - \( \forall X \in N, \sum_{y \in T^*} \sum_{X \rightarrow y} P(X) = 1 \)
- A grammar G generates a language model L.
  \[
  \sum_{y \in T^*} \sum_{X \rightarrow y} P(X) = 1
  \]
A PCFG

<table>
<thead>
<tr>
<th>Rule</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td>1.0</td>
</tr>
<tr>
<td>VP → V NP</td>
<td>0.6</td>
</tr>
<tr>
<td>VP → V NP PP</td>
<td>0.4</td>
</tr>
<tr>
<td>NP → NP NP</td>
<td>0.1</td>
</tr>
<tr>
<td>NP → NP PP</td>
<td>0.2</td>
</tr>
<tr>
<td>NP → N</td>
<td>0.7</td>
</tr>
<tr>
<td>PP → P NP</td>
<td>1.0</td>
</tr>
</tbody>
</table>

The probability of trees and strings

- P(t) – The probability of a tree t is the product of the probabilities of the rules used to generate it.
- P(s) – The probability of the string s is the sum of the probabilities of the trees which have that string as their yield

\[
P(s) = \sum_j P(s, t) \quad \text{where } t \text{ is a parse of } s
\]

```
Tree and String Probabilities
```

- s = people fish tanks with rods
- P(t₁) = 1.0 × 0.7 × 0.4 × 0.5 × 0.6 × 0.7 × 1.0 × 0.2 × 1.0 × 0.7 × 0.1
  = 0.0008232
- P(t₂) = 1.0 × 0.7 × 0.6 × 0.5 × 0.6 × 0.1 × 0.7 × 1.0 × 0.2 × 1.0 × 0.7 × 0.1
  = 0.00024696
- P(s) = P(t₁) + P(t₂)
  = 0.0008232 + 0.00024696
  = 0.00107016
Chomsky Normal Form

• All rules are of the form $X \to YZ$ or $X \to w$
    - $X, Y, Z \in N$ and $w \in T$
• A transformation to this form doesn’t change the weak generative capacity of a CFG
    - That is, it recognizes the same language
    - But maybe with different trees
• Empties and unaries are removed recursively
• $n$-ary rules are divided by introducing new nonterminals ($n > 2$)

A phrase structure grammar

| $S \to NP\ VP$  | $N \to$ people       |
| $VP \to V\ NP$ | $N \to$ fish         |
| $VP \to V\ NP\ PP$ | $N \to$ tanks       |
| $NP \to NP\ NP$ | $N \to$ rods         |
| $NP \to NP\ PP$ | $V \to$ people       |
| $NP \to N$     | $V \to$ fish         |
| $NP \to e$      | $V \to$ tanks        |
| $PP \to P\ NP$ | $P \to$ with         |

Chomsky Normal Form steps

| $S \to NP\ VP$  | $N \to$ people       |
| $VP \to V\ NP$ | $N \to$ fish         |
| $VP \to V\ NP\ PP$ | $N \to$ tanks       |
| $NP \to NP\ PP$ | $V \to$ people       |
| $NP \to NP\ PP$ | $V \to$ fish         |
| $NP \to PP$    | $V \to$ tanks        |
| $NP \to PP$    | $P \to$ with         |

Chomsky Normal Form steps

| $S \to NP\ VP$  | $N \to$ people       |
| $VP \to V\ NP$ | $N \to$ fish         |
| $VP \to V\ NP\ PP$ | $N \to$ tanks       |
| $NP \to NP\ PP$ | $V \to$ people       |
| $NP \to NP\ PP$ | $V \to$ fish         |
| $NP \to PP$    | $V \to$ tanks        |
| $NP \to PP$    | $P \to$ with         |
Chomsky Normal Form steps

\[
S \rightarrow NP \ VP \\
NP \rightarrow V \ NP \\
VP \rightarrow V \ NP \ PP \\
V \rightarrow people \\
V \rightarrow fish \\
V \rightarrow tanks \\
V \rightarrow rods \\
NP \rightarrow people \\
NP \rightarrow fish \\
NP \rightarrow tanks \\
NP \rightarrow rods \\
PP \rightarrow with \\
\]

Chomsky Normal Form steps

\[
S \rightarrow NP \ VP \\
VP \rightarrow V \ NP \\
V \rightarrow people \\
V \rightarrow fish \\
V \rightarrow tanks \\
V \rightarrow rods \\
NP \rightarrow NP \\
NP \rightarrow NP \\
PP \rightarrow with \\
\]

Chomsky Normal Form steps

\[
S \rightarrow NP \ VP \\
NP \rightarrow V \ NP \\
V \rightarrow people \\
V \rightarrow fish \\
V \rightarrow tanks \\
V \rightarrow rods \\
NP \rightarrow NP \\
NP \rightarrow NP \\
PP \rightarrow with \\
\]

Chomsky Normal Form

- You should think of this as a transformation for efficient parsing
- With some extra book-keeping in symbol names, you can even reconstruct the same trees with a detransform
- In practice full Chomsky Normal Form is a pain
  - Reconstructing n-aries is easy
  - Reconstructing unaries/empties is trickier
- **Binarization** is crucial for cubic time CFG parsing
- The rest isn't necessary; it just makes the algorithms cleaner and a bit quicker
An example: before binarization...

After binarization...

Treebank: empties and unaries

CKY Parsing

Constituency Parsing

Cocke-Kasami-Younger (CKY) Constituency Parsing
**The CKY algorithm (1960/1965)
--- extended to unaries**

```
function CKY(words, grammar) returns (most_probable_parse, prob)
score = new double[#(words)+1][#(words)+1][#(nonterms)]
back = new Pair[#(words)+1][#(words)+1][#(nonterms)]
for i = 0; i < #(words); i++
for A in nonterms
if A - words[i] in grammar
score[i][i+1][A] = P(A - words[i])
//handle unaries
while added
added = false
while added
added = false
for A, B in nonterms
prob = P(A - B) * score[begin][split][B];
if prob > score[begin][end][A]
score[begin][end][A] = prob
back[begin][end][A] = B
added = true
return buildTree(score, back)
```