Two views of linguistic structure:
1. Constituency (phrase structure)
   - Phrase structure organizes words into nested constituents.

   ![Phrase Structure Diagram]

1. 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 children] [about drugs].
     - John talked [about drugs] [to children].
     - *John talked drugs to the children about.
   - Substitution/expansion/pronoun:
     - I sat [on the box/right on top of the box/here].

Headed phrase structure
- Context-free grammar
  - VP → ... VB* ...
  - NP → ... NN* ...
  - ADVP → ... RB* ...
- SBAR(Q) → S | INF | SQ → ... NP VP ...

- Plus minor phrase types:
  - QP (quantifier phrase in NP), CONJP (multi-word constructions: as well as), INTJ (interjections), etc.
Two views of linguistic structure:
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

Phrase Chunking

- Find all non-recursive noun phrases (NPs) and verb phrases (VPs) in a sentence.
  - [NP] [NP | [VP ate | [NP the spaghetti | [PP with | [NP meatballs]]]
  - [NP He] [VP reckons | [NP the current account deficit | [VP will narrow | [PP to | [NP only 1.8 billion | [PP in | [NP September]]]

Phrase Chunking as Sequence Labeling

- Tag individual words with one of 3 tags
  - B (Begin) word starts new target phrase
  - I (Inside) word is part of target phrase but not the first word
  - O (Other) word is not part of target phrase
- Sample for NP chunking
  - He reckons the current account deficit will narrow to only 1.8 billion in September.

Evaluating Chunking

Per token accuracy does not evaluate finding correct full chunks. Instead use:

\[
\text{Precision} = \frac{\text{Number of correct chunks found}}{\text{Total number of chunks found}}
\]

\[
\text{Recall} = \frac{\text{Number of correct chunks found}}{\text{Total number of actual chunks}}
\]

Take harmonic mean to produce a single evaluation metric called F measure.

\[
F_1 = \frac{1}{\frac{1}{P} + \frac{1}{R} + \frac{2}{PR}}
\]

Current Chunking Results

- Best system for NP chunking: $F_1 = 96\%$
- Typical results for finding range of chunk types (CONLL 2000 shared task: NP, VP, PP, ADV, SBAR, ADJP) is $F_1 = 92\% - 94\%$
A Brief Parsing History

Pre 1990 (“Classical”) NLP Parsing
• Wrote symbolic grammar (CFG or often richer) and lexicon
  \[ S \rightarrow NP \, VP \]
  \[ NP \rightarrow (DT) \, NN \]
  \[ NP \rightarrow NN \, NNS \]
  \[ NP \rightarrow NNP \, VBP \]
  \[ VP \rightarrow V \, NP \]
• Used grammar systems to prove parses from words
• This scaled very badly. For sentence:
  Fed raises interest rates 0.5% in effort to control inflation
  • Minimal grammar:
    36 parses
  • Simple 10 rule grammar (with a lexicon):
    592 parses
  • Real-size broad-coverage grammar:
    millions of parses

Syntactic Parsing
• Produce the correct syntactic parse tree for a sentence.

Classical NLP Parsing: The problem and its solution
• Adding constraints 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

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
Two problems to solve for parsing:
1. Repeated work…

Two problems to solve for parsing:
2. Choosing the correct parse
   • How do we work out the correct attachment:
     • She saw the man with a telescope
     • Words are good predictors of attachment, even absent 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.

Statistical parsing applications
Statistical parsers are now robust and widely used in larger NLP applications:
• High precision question answering [Hocke and Harder SIGIR 2001]
• Improving biological named entity finding [Piatet et al. JAMIA 2004]
• Syntactically based sentence compression [Las and Wilks 2007]
• Extracting opinions about products [Hu et al. NAACL 2007]
• Improved interaction in computer games [Gonick and Ray 2006]
• Helping linguists find data [Resnik et al. BLS 2005]
• Source sentence analysis for machine translation [Hu et al. 2009]
• Relation extraction systems [Fundel et al. Bioinformatics 2006]

(Probabilistic) Context-Free Grammars
• CFG
• PCFG

A phrase structure grammar
S → NP VP
VP → V NP
VP → V NP PP
NP → NP VP
NP → NP PP
NP → N
NP → e
PP → P NP
people fish tanks
people fish with rods
N → people
N → fish
N → tanks
N → rods
V → people
V → fish
V → tanks
P → with
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 \((S \in N)\)
- \( R \) is a set of rules/productions of the form \(X \rightarrow g\)
  - \(X \in N\) and \(g \in (N \cup T)^*\)
- A grammar \(G\) generates a language \(L\).

Sentence Generation
- Sentences are generated by recursively rewriting the start symbol using the productions until only terminals symbols remain.

A phrase structure grammar

- In NLP
  - \( G = (T, C, N, S, L, R) \)
  - \( T \) is a set of terminal symbols
  - \( C \) is a set of preterminal symbols
  - \( N \) is a set of nonterminal symbols
  - \( S \) is the start symbol \((S \in N)\)
  - \( L \) is the lexicon, a set of items of the form \(X \rightarrow x\)
  - \( R \) is the grammar, a set of items of the form \(X \rightarrow g\)
    - \(X \in N\) and \(g \in (N \cup C)^*\)
  - By usual convention, \(S\) is the start symbol, but in statistical NLP, we usually have an extra node at the top (ROOT, TOP)
  - We usually write \(e\) for an empty sequence, rather than nothing

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 \((S \in N)\)
- \( R \) is a set of rules/productions of the form \(X \rightarrow \gamma\)
- \( P \) is a probability function
  - \( P : R \rightarrow [0,1] \)
  - \( \forall X \in N \), \(P(X \rightarrow \gamma) = 1 \)
  - \( \sum_{\gamma : X \rightarrow \gamma} P(X \rightarrow \gamma) = 1 \)
- A grammar \(G\) generates a language model \(L\).
### A PCFG

<table>
<thead>
<tr>
<th>Non-terminal</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.

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

\[
P(s) = \sum_t P(t)
\]

### Tree and String Probabilities

- \( s = \) people fish tanks with rods
- \( P(t_1) = 1.0 \times 0.7 \times 0.5 \times 0.6 \times 0.2 \times 0.7 \times 1.0 \times 0.2 \times 0.7 = 0.00024696 \)
- \( P(t_2) = 1.0 \times 0.7 \times 0.5 \times 0.6 \times 0.2 \times 0.7 \times 1.0 \times 0.2 \times 0.7 \times 0.7 = 0.00024696 \)
- \( P(t) = P(t_1) + P(t_2) = 0.00024696 + 0.00024696 = 0.00107016 \)

### Chomsky Normal Form

- All rules are of the form \( X \rightarrow YZ \) or \( X \rightarrow w \)
- A transformation to this form doesn’t change the generative capacity of a CFG
- That is, it recognizes the same language
- Empties and unaries are removed recursively
- \( n \)-ary rules are divided by introducing new nonterminals \( n > 2 \)

### A phrase structure grammar

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td></td>
</tr>
<tr>
<td>VP → V NP</td>
<td></td>
</tr>
<tr>
<td>VP → V NP PP</td>
<td></td>
</tr>
<tr>
<td>NP → NP NP</td>
<td></td>
</tr>
<tr>
<td>NP → NP PP</td>
<td></td>
</tr>
<tr>
<td>NP → N</td>
<td></td>
</tr>
<tr>
<td>NP → e</td>
<td></td>
</tr>
<tr>
<td>PP → P NP</td>
<td></td>
</tr>
</tbody>
</table>

### Verb attach

### Noun attach
Chomsky Normal Form steps

$S \rightarrow NP \ VP$
$NP \rightarrow N$
$NP \rightarrow V NP$
$NP \rightarrow V PP$
$NP \rightarrow N$
$VP \rightarrow P \ NP$
$VP \rightarrow P$

$N \rightarrow people$
$N \rightarrow fish$
$N \rightarrow tanks$
$N \rightarrow rods$
$V \rightarrow people$
$V \rightarrow fish$
$V \rightarrow tanks$
$P \rightarrow with$

Chomsky Normal Form steps

$S \rightarrow NP \ VP$
$NP \rightarrow N$
$NP \rightarrow V NP$
$NP \rightarrow V PP$
$NP \rightarrow N$
$VP \rightarrow P \ NP$
$VP \rightarrow P$

$N \rightarrow people$
$N \rightarrow fish$
$N \rightarrow tanks$
$N \rightarrow rods$
$V \rightarrow people$
$V \rightarrow fish$
$V \rightarrow tanks$
$P \rightarrow with$
Chomsky Normal Form

- You should think of this as a transformation for efficient parsing
- **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...

```
ROOT
   |  S
   |  NP
   |  VP
   |  N  
   |  V  
   |  NP  
   |  PP  
   |  P  
   |  NP  
   |  people
   |  fish
   |  table
   |  book
   |  rods
   |  with
   |  tanks
   |  fish

ROOT
```

After binarization on VP

```
ROOT
   |  S
   |  NP
   |  VP
   |  N
   |  V
   |  NP
   |  PP
   |  P
   |  NP
   |  people
   |  fish
   |  table
   |  book
   |  rods
   |  with
   |  tanks
   |  fish

Root
```

Parsing

- Given a string of terminals and a CFG, determine if the string can be generated by the CFG
- Also return a parse tree for the string
- Also return all possible parse trees for the string
- Must search space of derivations for one that derives the given string.
  - **Top-Down Parsing**: Start searching space of derivations for the start symbol.
  - **Bottom-up Parsing**: Start search space of reverse derivations from the terminal symbols in the string.

Parsing Example

```
S
   |  book that flight
   |  VP
   |  Nominal
   |  that
   |  Noun
   |  flight
```

Top Down Parsing

```
VP
   |  Premium
```

```
Top Down Parsing

Top Down Parsing

Top Down Parsing

Top Down Parsing

Top Down Parsing

Top Down Parsing
Top Down Parsing

Bottom Up Parsing

Noun
book that flight

Nominal
book that flight

Bottom Up Parsing
Bottom Up Parsing

Bottom Up Parsing

Bottom Up Parsing

Bottom Up Parsing

Bottom Up Parsing

Bottom Up Parsing

Bottom Up Parsing
**Bottom Up Parsing**

- Verb
- NP
- Nominal
- flight

---

**Top Down vs. Bottom Up**

- Top down never explores options that will not lead to a full parse, but can explore many options that never connect to the actual sentence.
- Bottom up never explores options that do not connect to the actual sentence but can explore options that can never lead to a full parse.
- Relative amounts of wasted search depend on how much the grammar branches in each direction.

---

**Dynamic Programming Parsing**

- To avoid extensive repeated work, must cache intermediate results, i.e. completed phrases.
- Caching (memorizing) is critical to obtaining a polynomial time parsing (recognition) algorithm for CFGs.
- Dynamic programming algorithms based on both top-down and bottom-up search can achieve $O(n^3)$ time where $n$ is the length of the input string.

---

**(Probabilistic) CKY Parsing**

---

**Constituency Parsing**

---

**Cocke-Kasami-Younger (CKY) Constituency Parsing**

---
**Viterbi (Max) Scores**

- **people**
  - NP: 0.35
  - V: 0.1
  - N: 0.5
- **fish**
  - NP: 0.14
  - V: 0.6
  - N: 0.2

**Extended CKY parsing**

- Unaries can be incorporated into the algorithm
- Messy, but doesn’t increase algorithmic complexity
- Empties can be incorporated
  - Doesn’t increase complexity; essentially like unaries
- Binarization is vital
  - Without binarization, you don’t get parsing cubic in the length of the sentence and in the number of nonterminals in the grammar

---

**The CKY algorithm (1960/1965)**

... extended to unaries

```java
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 in grammar
      score[i][i+1][A] = P(A > words[i])
  //handle unaries
  while added
    added = false
    for A, B in nonterms
      if score[i][i+1][B] > 0 && A > B in grammar
        prob = P(A > B) * score[i][i+1][B]
        if prob > score[i][i+1][A]
          score[i][i+1][A] = prob
          back[i][i+1][A] = B
          added = true
  return buildTree(score, back)
```

---

**The CKY algorithm (1960/1965)**

... extended to unaries

```java
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 in grammar
      score[i][i+1][A] = P(A > words[i])
  //handle unaries
  while added
    added = false
    for A, B in nonterms
      if score[i][i+1][B] > 0 && A > B in grammar
        prob = P(A > B) * score[i][i+1][B]
        if prob > score[i][i+1][A]
          score[i][i+1][A] = prob
          back[i][i+1][A] = B
          added = true
  return buildTree(score, back)
```