Syntax and Parsing

22 March 2021
Hexy planars garumphed by the snox three zamfirs ago.
Hexy planars garumphed by the snox three zamfirs ago.

What kind of planars are being discussed?
Hexy planars garumphed by the snox three zamfirs ago
Parsing in NLP

Annotating a sentence with a tree structure that describes its syntactic (grammatical) structure.

Generally, parsing requires or subsumes the task of part of speech tagging – annotating the words in a sentence with the syntactic roles that they play (what's a noun, what's a verb).
Why parse?

Question answering

When did *hexy planars garumph*?

Sentence compression / summarization

*Hexy planars garumped.*

Machine translation

*Planares hexos …*

Corpus linguistics

How often do people use *garumph* with an object vs without?
Part-of-speech tagging
# Parts of speech:
A traditional/informal view

<table>
<thead>
<tr>
<th>Part of Speech</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Noun</td>
<td>Names of things</td>
<td>boy, cat, truth</td>
</tr>
<tr>
<td>Verb</td>
<td>Action or state</td>
<td>become, hit</td>
</tr>
<tr>
<td>Pronoun</td>
<td>Used for noun</td>
<td>I, you, we</td>
</tr>
<tr>
<td>Adverb</td>
<td>Modifies V, Adj, Adv</td>
<td>sadly, very</td>
</tr>
<tr>
<td>Adjective</td>
<td>Modifies noun</td>
<td>happy, clever</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Joins things</td>
<td>and, but, while</td>
</tr>
<tr>
<td>Preposition</td>
<td>Relation of N</td>
<td>to, from, into</td>
</tr>
<tr>
<td>Interjection</td>
<td>An outcry</td>
<td>ouch, oh, alas, psst</td>
</tr>
</tbody>
</table>
Parts of speech:
The substitution test

The \{\textit{small, square, expensive, black, …}\} pot is on the stove.
# Parts of speech: The most common set for English

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Example</th>
<th>Tag</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>coordin. conjunction</td>
<td><em>and, but, or</em></td>
<td>SYM</td>
<td>symbol</td>
<td>+, %, &amp;</td>
</tr>
<tr>
<td>CD</td>
<td>cardinal number</td>
<td><em>one, two</em></td>
<td>TO</td>
<td>“to”</td>
<td>to</td>
</tr>
<tr>
<td>DT</td>
<td>determiner</td>
<td><em>a, the</em></td>
<td>UH</td>
<td>interjection</td>
<td>ah, oops</td>
</tr>
<tr>
<td>EX</td>
<td>existential ‘there’</td>
<td><em>there</em></td>
<td>VB</td>
<td>verb base form</td>
<td>eat</td>
</tr>
<tr>
<td>FW</td>
<td>foreign word</td>
<td><em>mea culpa</em></td>
<td>VBD</td>
<td>verb past tense</td>
<td>ate</td>
</tr>
<tr>
<td>IN</td>
<td>preposition/sub-conj</td>
<td><em>of, in, by</em></td>
<td>VBG</td>
<td>verb gerund</td>
<td>eating</td>
</tr>
<tr>
<td>JJ</td>
<td>adjective</td>
<td><em>yellow</em></td>
<td>VBN</td>
<td>verb past participle</td>
<td>eaten</td>
</tr>
<tr>
<td>JJR</td>
<td>adj., comparative</td>
<td><em>bigger</em></td>
<td>VBP</td>
<td>verb non-3sg pres</td>
<td><em>eat</em></td>
</tr>
<tr>
<td>JJS</td>
<td>adj., superlative</td>
<td><em>wildest</em></td>
<td>VBZ</td>
<td>verb 3sg pres</td>
<td><em>eats</em></td>
</tr>
<tr>
<td>LS</td>
<td>list item marker</td>
<td><em>1, 2, One</em></td>
<td>WDT</td>
<td>wh-determiner</td>
<td><em>which, that</em></td>
</tr>
<tr>
<td>MD</td>
<td>modal</td>
<td><em>can, should</em></td>
<td>WP</td>
<td>wh-pronoun</td>
<td><em>what, who</em></td>
</tr>
<tr>
<td>NN</td>
<td>noun, sing. or mass</td>
<td><em>llama</em></td>
<td>WP$</td>
<td>possessive wh-</td>
<td><em>whose</em></td>
</tr>
<tr>
<td>NNS</td>
<td>noun, plural</td>
<td><em>llamas</em></td>
<td>WRB</td>
<td>wh-adverb</td>
<td><em>how, where</em></td>
</tr>
<tr>
<td>NNP</td>
<td>proper noun, sing.</td>
<td><em>IBM</em></td>
<td>$</td>
<td>dollar sign</td>
<td>$</td>
</tr>
<tr>
<td>NNPS</td>
<td>proper noun, plural</td>
<td><em>Carolin</em></td>
<td>#</td>
<td>pound sign</td>
<td>#</td>
</tr>
<tr>
<td>PDT</td>
<td>predeterminer</td>
<td><em>all, both</em></td>
<td>“</td>
<td>left quote</td>
<td>‘ or “</td>
</tr>
<tr>
<td>POS</td>
<td>possessive ending</td>
<td><em>’s</em></td>
<td>”</td>
<td>right quote</td>
<td>’ or ”</td>
</tr>
<tr>
<td>PRP</td>
<td>personal pronoun</td>
<td><em>I, you, he</em></td>
<td>(</td>
<td>left parenthesis</td>
<td>[ (, {, &lt;</td>
</tr>
<tr>
<td>PRPS</td>
<td>possessive pronoun</td>
<td><em>your, one’s</em></td>
<td>)</td>
<td>right parenthesis</td>
<td>] ), ) &gt;</td>
</tr>
<tr>
<td>RB</td>
<td>adverb</td>
<td><em>quickly, never</em></td>
<td>,</td>
<td>comma</td>
<td>,</td>
</tr>
<tr>
<td>RBR</td>
<td>adverb, comparative</td>
<td><em>faster</em></td>
<td>.</td>
<td>sentence-final punc</td>
<td>. ! ?</td>
</tr>
<tr>
<td>RBS</td>
<td>adverb, superlative</td>
<td><em>fastest</em></td>
<td>:</td>
<td>mid-sentence punc</td>
<td>; ... -- -</td>
</tr>
</tbody>
</table>

*Figure 10.1* | Penn Treebank part-of-speech tags (including punctuation).
Part-of-speech tagging

Largely (but not perfectly) solved by simple machine-learning methods, included in typical parsing packages.
Constituency syntax
Sentences have parts, some of which appear to have subparts.

These groups of words that go together are called **constituents**.

*I saw [the man with the telescope].*

*I saw [the man] [with the telescope].*

*You [could not] go to her party.*

*You could [not go] to her party.*
Constituent types

For constituents, we usually name them as phrases based on the word that heads the constituent:

- *the man from Amherst* is a noun phrase (NP) because the head *man* is a noun
- *extremely clever* is an adjective phrase (AP) because the head *clever* is an adjective
- *down the river* is a prepositional phrase (PP) because the head *down* is a prep.
- *killed the rabbit* is a verb phrase (VP) because the head *killed* is a verb
Constituents and words

Note that a *word* is a constituent (albeit a little one).

Sometimes words also act as phrases:

*Joe grew potatoes.*

*Joe* and *potatoes* are both nouns and noun phrases.

Compare with:

*The man from Amherst grew beautiful russet potatoes.*
Some constituency tests

Linguists characterize constituents in a number of ways, including

where they occur (e.g., “NPs can occur before verbs”)

where they can move in variations of a sentence

   On September 17th, I’d like to fly from Atlanta to Denver
   I’d like to fly on September 17th from Atlanta to Denver
   I’d like to fly from Atlanta to Denver on September 17th

what parts can move and what parts can’t

   *On September I’d like to fly 17th from Atlanta to Denver

what they can be conjoined with

   I’d like to fly from Atlanta to Denver on September 17th and in the morning
Common constituents
Noun phrase (NP)

*Harry the Horse*

*a high-class* spot such as Mindy’s

*the Broadway* coppers

*the reason* he comes into the Hot Box

*they*

*three parties from Brooklyn*
Noun phrase (NP)

*Harry* [the Horse]

[a high-class *spot*] such as [Mindy’s]

the Broadway *coppers*

[the *reason*] he comes into [the Hot Box]

*they*

[three *parties*] from Brooklyn

*Adapted from J&M*
Verb phrase (VP)

prefer a morning flight

leave Boston in the morning

gave Delta $200

flew

would not eat Jell-O
Verb phrase (VP)

prefer [a morning flight]
leave [Boston] in [the morning]
gave [Delta] [$200]
flew

would not eat [Jell-O]
Prepositional phrase (PP)

to Seattle

on these flights

in Minneapolis

about the ground transportation in Chicago

of the round trip flight on United Airlines
Prepositional phrase (PP)

to [Seattle]
on [these flights]
in [Minneapolis]
about [[the ground transportation] [in [Chicago]]]
of [[the round trip flight] [on [United Airlines]]]
Constituency is recursive

I pet [the cat].

I pet [[the cat] [who bit [the dog]].

I pet [[the cat] [who bit [[the dog] [with the black spot]]]].
Context-free grammar
A *context-free grammar* (CFG) is the most common way of modeling constituency.

Also called *phrase structure grammar*.

And equivalent to Backus–Naur form (BNF).

First formalized by Chomsky in 1956.
A context-free grammar consists of

A finite set of *non-terminal symbols* (or *variables*), \( N \)

A *start symbol* \( S \in N \)

A set of *terminal symbols*, \( \Sigma \),

Distinct from \( N \)

Like the "alphabet" \( \Sigma \) in CMPU 240, but they can be full words

A set of production *rules* \( R \), each of the form \( A \rightarrow \beta \), where

The lefthand side \( A \) is a nonterminal from \( N \)

The righthand side \( \beta \) is a sequence of zero or more terminals and/or nonterminals: \( \beta \in (N \cup \Sigma)^* \)

A grammar is fully expressed as the tuple \((N, \Sigma, R, S)\)
An example CFG for a tiny bit of English

\[ G = (N, \Sigma, R, S) \]

\[ N = \{ S, NP, N, VP, Det, Noun, Verb, Aux \} \]

\[ \Sigma = \{ that, this, a, the, man, book, flight, meal, include, read, does \} \]

\[ R = \{ \]

\[ S \rightarrow NP \ VP \] \quad \text{Det} \rightarrow that \mid this \mid a \mid the

\[ S \rightarrow Aux \ NP \ VP \] \quad \text{Noun} \rightarrow book \mid flight \mid meal \mid man

\[ S \rightarrow VP \] \quad \text{Verb} \rightarrow book \mid include \mid read

\[ NP \rightarrow Det \ N \] \quad \text{Aux} \rightarrow does

\[ N \rightarrow Noun \]

\[ N \rightarrow Noun \ NN \]

\[ VP \rightarrow Verb \]

\[ VP \rightarrow Verb \ NP \]

\[ \} \]

\[ S = S \]
Application of grammar rewrite rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow NP \ VP )</td>
<td></td>
</tr>
<tr>
<td>( S \rightarrow Aux \ NP \ VP )</td>
<td></td>
</tr>
<tr>
<td>( S \rightarrow VP )</td>
<td></td>
</tr>
<tr>
<td>( NP \rightarrow Det \ N )</td>
<td></td>
</tr>
<tr>
<td>( N \rightarrow Noun )</td>
<td></td>
</tr>
<tr>
<td>( N \rightarrow Noun \ NN )</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>( Det \rightarrow that</td>
<td>this</td>
</tr>
<tr>
<td>( Noun \rightarrow book</td>
<td>flight</td>
</tr>
<tr>
<td>( Verb \rightarrow book</td>
<td>include</td>
</tr>
<tr>
<td>( Aux \rightarrow does )</td>
<td></td>
</tr>
</tbody>
</table>
Example phrase structure tree

The phrase-structure tree represents both the syntactic structure of the sentence and the *derivation* of the sentence under the grammar. E.g., \[ \text{VP} \rightarrow \text{Verb NP} \].
The first phrase-structure tree

Sentence
  NP
    The
    man
  VP
    V
      took
      the
    NP
      book

Chomsky, 1956
The first phrase-structure tree, updated

S

NP

Det
The

N
man

Verb
took

VP

Det
the

NP

N
book
CFGs can capture recursion

Example of seemingly endless recursion of embedded prepositional phrases:

\[
\begin{align*}
PP & \rightarrow \text{Prep NP} \\
NP & \rightarrow \text{Noun PP}
\end{align*}
\]

\[
[5 \text{ The mailman ate} \\
[NP \text{ his lunch}] \\
[PP \text{ with his friend}] \\
[PP \text{ from the cleaning staff}] \\
[PP \text{ of the building}] \\
[PP \text{ at the intersection}] \\
[PP \text{ on the north end}] \\
[PP \text{ of town}]]]]]]].
\]
Grammaticality

A CFG defines a *formal language* – the set of all sentences (strings of words) that can be derived by the grammar.

- Sentences in this set said to be *grammatical*.
- Sentences outside this set said to be *ungrammatical*.
Parsing
What is parsing?

Instead of running the grammar *forwards* to generate a sentence in the language, we want to run it *backwards* to find the structure of a sentence we’re given.

We want to find *all* structures matching an input string of words.

This can be done bottom-up (starting with the words) or top-down (starting with $S$)
While a grammar defines the relation between sentences and their syntactic structures, it doesn’t tell how to actually go about discovering the structure of a sentence.

For that, we need a *parsing algorithm*. 
The parsing algorithms we use for programming languages are very inefficient for natural languages.

Programming languages are designed so that there’s no ambiguity — not true for natural language!
Ambiguity
Ambiguity

**NP bracketing**

plastic cat food can cover

→ ? (plastic cat) (food can) cover

→ ? plastic (cat food can) cover

→ ? (plastic cat food) (can cover)

**Conjunctions and appositives**

…to my parents, Ayn Rand and God

→ ? (my parents), (Ayn Rand) and (God)

→ ? (my parents), (Ayn Rand and God)
When our parsers have difficulty finding the right parse, it’s worth remembering that people can too!

A *garden path sentence* is one where the sentence structure needs to be reanalyzed when the last word is encountered, e.g.,

*The horse raced past the barn fell.*

(the horse [that was] raced past the barn) fell.
To apply standard parsing algorithms to natural language, we need to do some form of \textit{backtracking} during the parse:

> When “stuck”, return to the most recent decision point.
CYK parsing
For efficient parsing, we want to use *dynamic programming* – store partial results in tables, letting us avoid repeated work on shared sub-problems.

This efficiently stores ambiguous structures with shared sub-parts.
CYK parsing

Reasonably efficient — $O(n^3)$ for $n$ words

Easy to code

Easily extendable to weighted case

Requires the grammar to be in Chomsky normal form:

Nonterminal rules are of the form $X \rightarrow YZ$

Terminal rules are of the form $X \rightarrow x$
CYK parsing: intuition

Consider the rule $D \rightarrow w$

Terminal (word) forms a constituent
Trivial to apply

Consider the rule $A \rightarrow B C$

If there is an $A$ somewhere in the input then there must be a $B$ followed by a $C$ in the input
First, precisely define span $[i, j]$
If $A$ spans from $i$ to $j$ in the input, then there must be some $k$ such that $i < k < j$
Easy to apply: we just need to try different values for $k$
**Input:**

sentence of length \( n+1 = (w_0, w_1, \ldots, w_n) \)

a CFG

Build a two-dimensional table with states of the form \([X, i, j]\)

“I can cover from words \( i \) through \( j \) with at least one tree rooted in \( X \)”

States contain backpointers of the form \( \{R, k\} \)

“To satisfy my state, I will use \( R (X \rightarrow Y Z) \) and states \([Y, i, k]\) and \([Z, k, j]\)”
**Initialize:** build \([X, i, i+1]\) for every lexical rule \(R = X \rightarrow w_i\); add backpointer \(\{R, ()\}\)

**Recursively:** build \([X, i, j]\) for every nonlexical rule \(R = X \rightarrow Y Z\) and pair of states \([Y, i, k], [Z, k, j]\) where \(i < k < j\); add backpointer \(\{R, k\}\)

If state is already built, just add backpointer

**Goal:** build \([X, 0, n+1]\) for start symbol \(X\)
time flies like an arrow

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like
<table>
<thead>
<tr>
<th>0,1</th>
<th>0,2</th>
<th>0,3</th>
<th>0,4</th>
<th>0,5</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>flies</td>
<td>like</td>
<td>an</td>
<td>arrow</td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like
time flies like an arrow

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like
time flies like an arrow
<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[0,2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[0,3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[0,4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[0,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like

Length 4
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
</tbody>
</table>

**Grammar Rules:**

- S → NP VP
- NP → DT NN
- NP → time | fruit
- NP → NN NNS
- VP → VBP NP
- VP → flies
- VP → VP PP
- PP → IN NP
- DT → a | an
- NN → time | fruit | arrow | banana
- NNS → flies
- VBP → like
- IN → like

Length 5
Seed length-1 cells
with lexical coverage
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
</tr>
<tr>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
**VP → flies**
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
**NNS → flies**
VBP → like
IN → like

Seed length-1 cells
with lexical coverage
time flies like an arrow

Seed length-1 cells with lexical coverage
Seed length-1 cells with lexical coverage
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NN</td>
<td>VP</td>
<td>NNS</td>
<td>VBP</td>
</tr>
<tr>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like

Seed length-1 cells with lexical coverage
Now check length-2 spans.
Match rules to subspans

<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NN</td>
<td>[0,1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
</tr>
<tr>
<td>S →  NP VP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP → DT NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP → time</td>
<td>fruit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP → NN NNS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VP → VBP NP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VP → flies</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VP → VP PP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PP → IN NP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT → a</td>
<td>an</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN → time</td>
<td>fruit</td>
<td>arrow</td>
<td>banana</td>
<td></td>
</tr>
<tr>
<td>NNS → flies</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VBP → like</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN → like</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now check length-2 spans.
Match rules to subspans
Two new states!
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
</tbody>
</table>

S → NP VP  
NP → DT NN  
NP → time | fruit  
NP → NN NNS  
VP → VBP NP  
VP → flies  
VP → VP PP  
PP → IN NP  
DT → a | an  
NN → time | fruit | arrow | banana  
NNS → flies  
VBP → like  
IN → like  

Annotate with backpointer for future reconstruction (not part of state)
time flies like an arrow
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td></td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td>[0,1]</td>
<td></td>
<td>[0,2]</td>
<td>[0,3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NNS</td>
<td>[0,5]</td>
<td></td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like

No matching rule
New state!
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1]</td>
<td>NP</td>
<td>NN</td>
<td>[0,2]</td>
<td>S</td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like

Check every split point for rule
<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1]</td>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,2]</td>
<td>NP</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,3]</td>
<td>NN</td>
<td>NNS</td>
<td>IN</td>
<td>VBP</td>
<td></td>
</tr>
<tr>
<td>[0,4]</td>
<td></td>
<td></td>
<td>DT</td>
<td>NP</td>
<td></td>
</tr>
<tr>
<td>[0,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like
time flies like an arrow

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like

No matching rule
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VP</td>
<td>NNS</td>
<td>VBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1,2]</td>
<td>[1,3]</td>
<td>[1,4]</td>
<td>[1,5]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td></td>
<td>DT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2,3]</td>
<td>[2,4]</td>
<td>[2,5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VBP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3,4]</td>
<td>[3,5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like

(Skipping)
<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>NP</td>
<td>VP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>time</td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>fruit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $S \rightarrow NP \ VP$
- $NP \rightarrow DT \ NN$
- $NP \rightarrow \text{time} \ | \ \text{fruit}$
- $NP \rightarrow \text{NN} \ \text{NNS}$
- $VP \rightarrow \text{VBP} \ \text{NP}$
- $VP \rightarrow \text{flies}$
- $VP \rightarrow \text{VP} \ \text{PP}$
- $PP \rightarrow \text{IN} \ \text{NP}$
- $DT \rightarrow a \ | \ \text{an}$
- $NN \rightarrow \text{time} \ | \ \text{fruit} \ | \ \text{arrow} \ | \ \text{banana}$
- $\text{NNS} \rightarrow \text{flies}$
- $\text{VBP} \rightarrow \text{like}$
- $\text{IN} \rightarrow \text{like}$

Two new states!
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td></td>
<td>VBP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>NNS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S → NP VP  
NP → DT NN  
NP → time berry  
NP → NN NNS  
VP → VBP NP  
VP → flies  
VP → VP PP  
PP → IN NP  
DT → a berry  
NN → time berry arrow banana  
NNS → flies  
VBP → like  
IN → like  

Empty cell
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td>NNS</td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like

(Skipping)
<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td></td>
<td>S</td>
<td></td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>NN</td>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
<tr>
<td>S</td>
<td>NP</td>
<td>VP</td>
<td></td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>NP</td>
<td>DT</td>
<td>NN</td>
<td></td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>NP</td>
<td>time</td>
<td>fruit</td>
<td></td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>NP</td>
<td>NN</td>
<td>NNS</td>
<td></td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>VP</td>
<td>VBP</td>
<td>NP</td>
<td></td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>VP</td>
<td>flies</td>
<td></td>
<td></td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>VP</td>
<td>VP</td>
<td>PP</td>
<td></td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>PP</td>
<td>IN</td>
<td>NP</td>
<td></td>
<td></td>
<td>NP</td>
</tr>
<tr>
<td>DT</td>
<td>a</td>
<td>an</td>
<td></td>
<td></td>
<td>NN</td>
</tr>
<tr>
<td>NN</td>
<td>time</td>
<td>fruit</td>
<td>arrow</td>
<td>banana</td>
<td></td>
</tr>
<tr>
<td>NNS</td>
<td>flies</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VBP</td>
<td>like</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>like</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New state!
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>fruit</td>
<td>like</td>
<td>an</td>
<td>arrow</td>
</tr>
</tbody>
</table>

- **S** → NP VP
- NP → DT NN
- NP → time | fruit
- NP → NN NNS
- VP → VBP NP
- VP → flies
- VP → VP PP
- PP → IN NP
- DT → a | an
- NN → time | fruit | arrow | banana
- NNS → flies
- VBP → like
- IN → like

Empty Cell
<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1]</td>
<td>NP</td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,2]</td>
<td>S</td>
<td></td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,3]</td>
<td>VP</td>
<td>VBP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grammar rules:
- S → NP VP
- NP → DT NN
- NP → time | fruit
- NP → NN NNS
- VP → VBP NP
- VP → flies
- VP → VP PP
- PP → IN NP
- DT → a | an
- NN → time | fruit | arrow | banana
- NNS → flies
- VBP → like
- IN → like

Empty Cell
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NP</strong></td>
<td><strong>S</strong></td>
<td><strong>VP</strong></td>
<td><strong>NP</strong></td>
<td><strong>S</strong></td>
</tr>
<tr>
<td><strong>NN</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
</tbody>
</table>

Production rules:

- `S → NP VP`
- `NP → DT NN`
- `NP → time | fruit`
- `NP → NN NNS`
- `VP → VBP NP`
- `VP → flies`
- `VP → VP PP`
- `PP → IN NP`
- `DT → a | an`
- `NN → time | fruit | arrow | banana`
- `NNS → flies`
- `VBP → like`
- `IN → like`

New state!
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NP</strong></td>
<td><strong>S</strong></td>
<td><strong>VP</strong></td>
<td><strong>an</strong></td>
<td><strong>S</strong></td>
</tr>
</tbody>
</table>

- S → NP VP
- NP → DT NN
- NP → time | fruit
- NP → NN NNS
- VP → VBP NP
- VP → flies
- VP → VP PP
- PP → IN NP
- DT → a | an
- NN → time | fruit | arrow | banana
- NNS → flies
- VBP → like
- IN → like

Alternate (no new state)
time flies like an arrow
time flies like an arrow

S → NP VP
NP → DT NN
NP → time | fruit
NP → NN NNS
VP → VBP NP
VP → flies
VP → VP PP
PP → IN NP
DT → a | an
NN → time | fruit | arrow | banana
NNS → flies
VBP → like
IN → like
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td>IN</td>
<td>VP</td>
</tr>
<tr>
<td>NN</td>
<td>NP</td>
<td>NNS</td>
<td>VBP</td>
<td>VP</td>
</tr>
<tr>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
<tr>
<td>[1,2]</td>
<td>[1,3]</td>
<td>[1,4]</td>
<td>[1,5]</td>
<td></td>
</tr>
</tbody>
</table>

Start at top of the chart
Follow backpointers to build tree

time flies like an arrow
NP NN S NP VP NNS IN VBP VP PP VP NP NN
time flies like an arrow
time flies like an arrow
time flies like an arrow

NP NN [0,1] S NP [0,2] S [0,5]
VP NNS [1,2] VP [1,5]
IN VBP [2,3] PP VP [2,5]
DT [2,4] NP [3,5]
NN [4,5]

(could have used $S \rightarrow NP \ VP$, 2 to generate the other tree)

Backpointer: $S \rightarrow NP \ VP$, 1
time flies like an arrow
<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NN</td>
<td>S</td>
<td>NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VP</td>
<td>NNS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>VBP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Backpointer: VP → VP PP ,2
<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1]</td>
<td>NP</td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,2]</td>
<td>S</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1,2]</td>
<td>VP</td>
<td>NNS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1,3]</td>
<td>IN</td>
<td>VBP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2,3]</td>
<td>DT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3,4]</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4,5]</td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Backpointer: VP → flies
time flies like an arrow

Backpointer: PP → IN NP, 3
time flies like an arrow

<table>
<thead>
<tr>
<th></th>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td>S</td>
<td>VP</td>
</tr>
<tr>
<td>1</td>
<td>NP</td>
<td>NNS</td>
<td>VP</td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>2</td>
<td>VP</td>
<td>IN</td>
<td>VBP</td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>3</td>
<td>VP</td>
<td>PP</td>
<td>DT</td>
<td></td>
<td>NP</td>
</tr>
<tr>
<td>4</td>
<td>VP</td>
<td>PP</td>
<td>DT</td>
<td></td>
<td>NP</td>
</tr>
</tbody>
</table>

Backpointer: IN → like
<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td>NP</td>
<td>NNS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Backpointer: NP→DT NN ,4
```
time flies like an arrow.
time flies like an arrow

<table>
<thead>
<tr>
<th>time</th>
<th>flies</th>
<th>like</th>
<th>an</th>
<th>arrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>S</td>
<td>VP</td>
<td>NNS</td>
<td>S</td>
</tr>
<tr>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[1,2]</td>
<td>[1,3]</td>
<td>[0,5]</td>
</tr>
<tr>
<td>VP</td>
<td>IN</td>
<td>VBP</td>
<td></td>
<td>VP</td>
</tr>
<tr>
<td>[1,2]</td>
<td>[2,3]</td>
<td>[2,4]</td>
<td>[2,5]</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>DT</td>
<td>NP</td>
<td></td>
<td>NP</td>
</tr>
<tr>
<td>[1,3]</td>
<td>[3,4]</td>
<td>[3,5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Backpointer: NN → arrow
Chart parsing
CYK is a *bottom-up* chart parsing algorithm.

We can also do a chart parse *top-down*.

But we'll skip the details!
**NLTK ChartParser**

Constructed from a grammar and a list of chart rules (strategy)

Two ready-made strategies:

- `TopDownChartParser` (top-down)
- `BottomUpChartParser` (bottom-up)

You can also create your own!
from nltk import CFG

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

parser = nltk.BottomUpChartParser(grammar)

sent = "old men and women".split()

for tree in parser.chart_parse(sent):
    print(tree)
nltk.parse.chart.demo(
    2,
    print_times=False,
    trace=1,
    sent="I saw John with a dog",
    numparses=2
)
The Penn Treebank and grammar induction
Where do CFGs come from?

We can write them by hand, using linguistic expertise.

But why not read them off of real trees?
S → NP VP.
S → NP VP.
NP → NP, ADJP,
S → NP VP .
NP → NP , ADJP ,
VP → MD VP
S → NP VP.
NP → NP, ADJP,
VP → MD VP
VP → VB NP PP NP

NP: Pierre Vinken
ADJP: 61 years old
MD: will join the board as a nonexecutive director
VP: MD VP
NP: Nov. 29
S → NP VP .
NP → NP , ADJP ,
VP → MD VP
VP → VB NP PP NP
...
NNP → Pierre
NNP → Vinken
...
The standard approach

Build a corpus of annotated sentences, called a treebank.

Extract rules from the treebank.

Optionally, use statistical models to generalize the rules.
The *Penn Treebank* (Marcus et al., 1993) was a project that aimed to investigate the syntactic phenomena that naturally occur.

40,000 sentences (a million words of English) from the Wall Street Journal were annotated with syntactic structure.

$1/word

Since then, treebanks have been created for many other languages.
Example from the Penn Treebank
Lisp-style encoding in the Penn Treebank

(S
 (NP-SBJ
  (NP (NNP Pierre) (NNP Vinken) )
  (, ,)
  (ADJP
   (NP (CD 61) (NNS years) )
   (JJ old) )
  (, ,) )
 (VP (MD will)
  (VP (VB join)
   (NP (DT the) (NN board) )
   (PP-CLR (IN as)
    (NP (DT a) (JJ nonexecutive) (NN director) ))
   (NP-TMP (NNP Nov.) (CD 29) )))
 (. .) )
More phrase types, from the PTB manual

Clauses

**S** - simple declarative clause, i.e. one that is not introduced by a (possible empty) subordinating conjunction or a wh-word and that does not exhibit subject-verb inversion.

**SBAR** - Clause introduced by a (possibly empty) subordinating conjunction.

**SBARQ** - Direct question introduced by a wh-word or a wh-phrase. Indirect questions and relative clauses should be bracketed as SBAR, not SBARQ.

**SINV** - Inverted declarative sentence, i.e. one in which the subject follows the tensed verb or modal.

**SQ** - Inverted yes/no question, or main clause of a wh-question, following the wh-phrase in SBARQ.

Phrases

**ADJP** - Adjective Phrase.

**ADVP** - Adverb Phrase.

**CONJP** - Conjunction Phrase.

**FRAG** - Fragment.

**INTJ** - Interjection. Corresponds approximately to the part-of-speech tag UH.

**LST** - List marker. Includes surrounding punctuation.

**NAC** - Not a Constituent; used to show the scope of certain prenominal modifiers within an NP.

**NP** - Noun Phrase.

**NX** - Used within certain complex NPs to mark the head of the NP. Corresponds very roughly to N-bar level but used quite differently.

**PP** - Prepositional Phrase.

**PRN** - Parenthetical.

**PRT** - Particle. Category for words that should be tagged RP.

**QP** - Quantifier Phrase (i.e. complex measure/amount phrase); used within NP.

**RRC** - Reduced Relative Clause.

**UCP** - Unlike Coordinated Phrase.

**VP** - Verb Phrase.

**WHADJP** - Wh-adjective Phrase. Adjectival phrase containing a wh-adverb, as in how hot.

**WHAVP** - Wh-adverb Phrase. Introduces a clause with an NP gap. May be null (containing the 0 complementizer) or lexical, containing a wh-adverb such as how or why.

**WHNP** - Wh-noun Phrase. Introduces a clause with an NP gap. May be null (containing the 0 complementizer) or lexical, containing some wh-word, e.g. who, which book, whose daughter, none of which, or how many leopards.

**WHPP** - Wh-prepositional Phrase. Prepositional phrase containing a wh-noun phrase (such as of which or by whose authority) that either introduces a PP gap or is contained by a WHNP.

**X** - Unknown, uncertain, or unbracketable. X is often used for bracketing typos and in bracketing the...the-constructions.
Why make the Treebank?

Before this it was common for linguists to declare what constituted a legitimate sentence structure in a language (this is a gross simplification).

The Treebank provides a documentation of how (a very specific subset of) people actually structure English.

Constructed with this goal: given a new sentence, can we automatically put the tree on top? (i.e. syntactic parsing)
How to use the treebank
How to use the treebank

Evaluation:

I’ll give you some of the sentences from the treebank (without trees).
You give me the trees your computer parser produced.
I’ll tell you how close you were to the human trees.
How to use the treebank

Evaluation:

I’ll give you some of the sentences from the treebank (without trees).
You give me the trees your computer parser produced.
I’ll tell you how close you were to the human trees.

Training:

Use the treebank trees as input to some data-driven parsing algorithm
(just don’t use the same trees to evaluate)
Standard split: sections 2–21 for training, section 23 for test
Some Penn Treebank rules with counts

40717 PP → IN NP
33803 S → NP-SBJ VP
22513 NP-SBJ → -NONE-
21877 NP → NP PP
20740 NP → DT NN
14153 S → NP-SBJ VP
12922 VP → TO VP
11881 PP-LOC → IN NP
11467 NP-SBJ → PRP
11378 NP → -NONE-
11291 NP → NN

989 VP → VBG S
985 NP-SBJ → NN
983 PP-MNR → IN NP
983 NP-SBJ → DT
969 VP → VBN VP
100 VP → VBD PP-PRD
100 PRN: NP:
100 NP → DT JJS
100 NP-CLR → NN
99 NP-Sbj-1 → DT NNP

98 VP → VBN NP PP-DIR
98 VP → VBD PP-TMP
98 PP-TMP → VBG NP
97 VP → VBD ADVP-TMP VP

10 WHNP-1 → WRB JJ
10 VP → VP CC VP PP-TMP
10 VP → VP CC VP ADVP-MNR
10 VP → VBZ S, SBAR-ADV
10 VP → VBZ S ADVP-TMP
Probabilistic parsing
Because language is ambiguous, our parsers may find multiple possible parse trees for a sentence.

How do we know which is best?

We need a weighted grammar.
Weighted grammars

Consider *give*

Requires

- a direct object (the thing being given) and
- an indirect object (the recipient)

Prepositional dative form

- indirect object appears last

*John gave the book to Mary.*

Double object form

- indirect object comes first

*John gave Mary the book.*
Give in the Penn Treebank

gave NP: the chefs / NP: a standing ovation
give NP: advertisers / NP: discounts for maintaining or increasing ad sp...  
give NP: it / PP-DTV: to the politicians
gave NP: them / NP: similar help
give NP: only French history questions / PP-DTV: to students in a Europe...
give NP: federal judges / NP: a raise
give NP: consumers / NP: the straight scoop on the U.S. waste crisis
gave NP: Mitsui / NP: access to a high-tech medical product
give NP: Mitsubishi / NP: a window on the U.S. glass industry
give NP: much thought / PP-DTV: to the rates she was receiving, nor to ...
give NP: your Foster Savings Institution / NP: the gift of hope and free...
give NP: market operators / NP: the authority to suspend trading in futu...
gave NP: quick approval / PP-DTV: to $3.18 billion in supplemental appr...
give NP: the Transportation Department / NP: up to 50 days to review any...
give NP: the president / NP: such power
give NP: me / NP: the heebie-jeebies
give NP: holders / NP: the right, but not the obligation, to buy a cal...
gave NP: Mr. Thomas / NP: only a ``qualified'' rating, rather than ``...
Observations

Strong tendency for the shortest complement to appear first

But that doesn't account for

give NP: federal judges / NP: a raise
Animacy playing a role?

There are a large number of contributing factors

Surveyed by Bresnan & Hay, 2006
To capture these tendencies, we can add weights (probabilities) to productions of a grammar, giving us a *probabilistic context-free grammar* (PCFG).
Probabilistic CFGs
A **probabilistic context-free grammar** consists of

A finite set of *non-terminal symbols* (or *variables*), $N$

A *start symbol* $S \in N$

A set of *terminal symbols*, $\Sigma$,

Distinct from $N$

Like the “alphabet” $\Sigma$ in CMPU 240, but they can be full words

A set of production *rules* $R$, each of the form $A \rightarrow \beta$, where

The lefthand side $A$ is a nonterminal from $N$

The righthand side $\beta$ is a sequence of zero or more terminals and/or nonterminals: $\beta \in (N \cup \Sigma)^*$

For each $A \in N$, a *probability distribution* over the rules where $A$ is the lefthand side, $p(*) \mid A)$
Defining a PCFG

```python
from nltk import PCFG
grammar = PCFG.fromstring(""

S     ->  NP VP               [1.0]
VP    ->  TV NP               [0.4]
VP    ->  IV                  [0.3]
VP    ->  DatV NP NP          [0.3]
TV    ->  'saw'               [1.0]
IV    ->  'ate'               [1.0]
DatV  ->  'gave'              [1.0]
NP    ->  'telescopes'        [0.8]
NP    ->  'Jack'              [0.2]

"")
```
PCFG parsers
PCFG parsers in NLTK

A* parser: nltk.parse.ViterbiParser

Uses Viterbi-style dynamic programming to find the single most likely parse for a given text.
When there are multiple possible parses for a subtree, it discards all but the most likely parse.

Bottom-up chart parser: nltk.parse.pchart

Maintains a queue of edges.
Added to chart one at a time.
Ordering of queue based on probabilities associated with the edges.
Expands more likely edges first.
Different queue orderings used to implement different search strategies.


A* parser

Iteratively fills in a most likely constituents table

Records the most likely tree for each span and node value

Returns the entry for the most likely constituent that spans the whole text

Uses the NLTK ViterbiParser

```python
>>> grammar = PCFG.fromstring(''
    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.parse.ViterbiParser(grammar)
>>> sent = 'old men and women'.split()
>>> for tree in print viterbi_parser.parse(sent):
    print(tree)
    (NP (JJ old) (NNS (NNS men) (CC and) (NNS women))) (p=0.000864)
```
Example

*I saw the man with the telescope*

<table>
<thead>
<tr>
<th>Span</th>
<th>Node</th>
<th>Tree</th>
<th>Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0:1]</td>
<td>NP</td>
<td>(NP I)</td>
<td>0.15</td>
</tr>
<tr>
<td>[6:7]</td>
<td>NP</td>
<td>(NN telescope)</td>
<td>0.5</td>
</tr>
<tr>
<td>[5:7]</td>
<td>NP</td>
<td>(NP the telescope)</td>
<td>0.2</td>
</tr>
<tr>
<td>[4:7]</td>
<td>PP</td>
<td>(PP with (NP the telescope))</td>
<td>0.122</td>
</tr>
<tr>
<td>[0:4]</td>
<td>S</td>
<td>(S (NP I) (VP saw (NP the man)))</td>
<td>0.01365</td>
</tr>
<tr>
<td>[0:7]</td>
<td>S</td>
<td>(S (NP I) (VP saw (NP (NP the man) (PP with (NP the telescope))))</td>
<td>0.0004163250</td>
</tr>
</tbody>
</table>

Two possible constituents cover the span [0:7] and have VP node values

“saw the man, who has the telescope”

(VP saw
  (NP (NP the man)
    (PP with (NP the telescope))))

“used the telescope to see the man”

(VP saw
  (NP the man)
  (PP with (NP the telescope)))

First has a higher probability, so parser discards second one
PCFG induction
We can estimate the probabilities for rules by reading training trees, like in the Penn Treebank.

Simplest method – as with other problems we’ve seen – is *maximum likelihood estimation*.

E.g., probability of $ VP \rightarrow V \ NP \ PP = p(V, NP, PP | VP) $  

$$ P(V, NP, PP | VP = \frac{\text{count}(VP \rightarrow V \ NP \ PP)}{\text{count}(VP)} $$

Fancy name for count and divide
NLTK: Induce grammar from Penn Treebank

```python
import nltk
from nltk.corpus import treebank
from nltk.grammar import CFG, Nonterminal, induce_pcfg

S = Nonterminal('S')
productions = treebank.parsed_sents()[0].productions()

mini_grammar = induce_pcfg(S, productions)
```
Problems with PCFGs

Using the approach just described ("vanilla PCFG"), a CYK parser scores about 73% F₁.

State of the art parsers are in the mid-90s.

Why so bad?

- Rules are too specific
- Rules are not specific enough
Too specific
Too specific

If we see this tree:
Too specific

If we see this tree:

We get this rule: \( NP \rightarrow DT \ JJS \ NN \ NN \ NN \ PP \)
Too specific

If we see this tree:

We get this rule: $\text{NP} \rightarrow \text{DT JJS NN NN NN PP}$

But we can’t parse this tree:
Too specific

If we see this tree:

We get this rule: \[ NP \rightarrow DT \ JJS \ NN \ NN \ NN \ PP \]

But we can’t parse this tree:

without this rule: \[ NP \rightarrow DT \ JJS \ NN \ PP \]
Not specific enough
Not specific enough

These derivations only differ by the highlighted rules
Not specific enough

These derivations only differ by the highlighted rules

If $p(VP \rightarrow VP PP) > p(NP \rightarrow NP PP)$ then the left derivation wins. Else, the right derivation
Not specific enough

These derivations only differ by the highlighted rules

If \( p(\text{VP} \rightarrow \text{VP PP}) > p(\text{NP} \rightarrow \text{NP PP}) \) then the left derivation wins. Else, the right derivation

The words (particularly “into”) are not considered. PPs with “into” are much more likely to attach to VP than NP.
A key idea: headedness
Head lexicalization
Head lexicalization

VP/dumped → VP/dumped PP/into
Head lexicalization

VP/dumped → VP/dumped PP/into
more likely than
Head lexicalization

VP/dumped → VP/dumped PP/into

more likely than

NP/sacks → NP/sacks PP/into
Sparseness and the Penn Treebank

Penn Treebank

1 million words of parsed English WSJ

Has been a key resource (because of the widespread reliance on supervised learning)

But 1 million words is like *nothing*:

- 965,000 constituents, but only 66 WHADJP, of which only 6 aren’t *how much* or *how many*, but there is an infinite space of these, e.g.,
  *how clever/original/incompetent* *(at risk assessment and evaluation)*

Many of the probabilities that you would like to compute, you can’t compute
The state of the art
As with most problems in NLP, currently (approx. last five years) the best parsers are designed using neural networks.

The methods used are quite new and relatively complicated; we won’t go into them (at least today).
Parser demo

Allen AI constituency parser

https://demo.allennlp.org/constituency-parsing
Ok, but I just need a good constituent parse – what do I use?

In 2021, my answer is the Berkeley Neural Parser (benepar).

https://github.com/nikitakit/self-attentive-parser
Acknowledgments

The lecture incorporates material from:

Sam Bowman, New York University
Nancy Ide, Vassar College
Daniel Jurafsky and James Martin, *Speech and Language Processing*
Jon May, University of Southern California
Noah Smith, University of Washington