Dependency Parsing
And Other Grammar Formalisms

Yejin Choi - University of Washington
Dependency Grammar

For each word, find one parent.

Child            Parent

A child is **dependent** on the parent.
- A child is an **argument** of the parent.
- A child **modifies** the parent.

I     shot     an     elephant
For each word, find one parent.

Child               Parent

A child is dependent on the parent.
- A child is an argument of the parent.
- A child modifies the parent.

I shot an elephant in my pajamas
For each word, find one parent.

Child: A child is dependent on the parent.
- A child is an argument of the parent.
- A child modifies the parent.
I shot an elephant in my pajamas yesterday.
I shot an elephant in my pajamas

Typed Dependencies:

nsubj(shot-2, i-1)
root(ROOT-0, shot-2)
det(elephant-4, an-3)
dobj(shot-2, elephant-4)

prep(shot-2, in-5)
poss(pajamas-7, my-6)
pobj(in-5, pajamas-7)
Naïve CKY Parsing

\[ O(n^5N^3) \text{ if } N \text{ nonterminals} \]

\[ O(n^5) \] combinations

slides from Eisner & Smith
Eisner Algorithm (Eisner & Satta, 1999)

This happens only once as the very final step

Without adding a dependency arc

When adding a dependency arc (head is higher)
Eisner Algorithm (Eisner & Satta, 1999)

A triangle is a head with some left (or right) subtrees.

One trapezoid per dependency.

It takes two to tango.
Eisner Algorithm (Eisner & Satta, 1999)

- $O(n)$ combinations
- $O(n^3)$ combinations
- $O(n^3)$ combinations

Gives $O(n^3)$ dependency grammar parsing
Eisner Algorithm

- **Base case:**
  \[ \forall t \in \{\sqsubseteq, \sqsupset, \sqsubset, \sqsupseteq\}, \ \pi(i, i, t) = 0 \]

- **Recursion:**
  \[
  \pi(i, j, \sqsubseteq) = \max_{i \leq k < j} \left( \pi(i, k, \sqsupset) + \pi(k + 1, j, \sqsubseteq) + \phi(w_j, w_i) \right)
  \]
  \[
  \pi(i, j, \sqsupset) = \max_{i \leq k < j} \left( \pi(i, k, \sqsupset) + \pi(k + 1, j, \sqsubset) + \phi(w_i, w_j) \right)
  \]
  \[
  \pi(i, j, \sqsubset) = \max_{i \leq k < j} \left( \pi(i, k, \sqsubset) + \pi(k + 1, j, \sqsubseteq) \right)
  \]
  \[
  \pi(i, j, \sqsupseteq) = \max_{i \leq k < j} \left( \pi(i, k, \sqsupseteq) + \pi(k + 1, j, \sqsupset) \right)
  \]

- **Final case:**
  \[
  \pi(1, n, \sqsubset \sqsupseteq) = \max_{1 \leq k < n} \left( \pi(1, k, \sqsubset) + \pi(k + 1, n, \sqsupset) \right)
  \]
CFG vs Dependency Parse

- **CFG** focuses on "constituency" (i.e., phrasal/clausal structure)
- **Dependency** focuses on "head" relations.

- CFG includes non-terminals. CFG edges are not typed.
- No non-terminals for dependency trees. Instead, dependency trees provide "dependency types" on edges.

- Dependency types encode "grammatical roles" like
  - nsubj -- nominal subject
  - dobj – direct object
  - pobj – prepositional object
  - nsubjpass – nominal subject in a passive voice
Can we get “heads” from CFG trees?
   Yes. In fact, modern statistical parsers based on CFGs use hand-written “head rules” to assign “heads” to all nodes.

Can we get constituents from dependency trees?
   Yes, with some efforts.

Can we transform CFG trees to dependency parse trees?
   Yes, and transformation software exists. (stanford toolkit based on [de Marneffe et al. LREC 2006])

Can we transform dependency trees to CFG trees?
   Mostly yes, but (1) dependency parse can capture non-projective dependencies, while CFG cannot, and (2) people rarely do this in practice
Both are context-free.
Both are used frequently today, but dependency parsers are more recently popular.

CKY Parsing algorithm:
- O \(N^3\) using CKY & unlexicalized grammar
- O \(N^5\) using CKY & lexicalized grammar (O\(N^4\) also possible)

Dependency parsing algorithm:
- O \(N^5\) using naïve CKY
- O \(N^3\) using Eisner algorithm
- O \(N^2\) based on minimum directed spanning tree algorithm (arborescence algorithm, aka, Edmond-Chu-Liu algorithm – see edmond.pdf)

Linear-time O \(N\) Incremental parsing (shift-reduce parsing) possible for both grammar formalisms
Non Projective Dependencies

- Mr. Tomash will remain as a director emeritus.

- A hearing is scheduled on the issue today.
Non Projective Dependencies

- Projective dependencies: when the tree edges are drawn directly on a sentence, it forms a tree (without a cycle), and there is no crossing edge.

- Projective Dependency:
- Eg:

  Mr. Tomash will remain as a director emeritus.
Non Projective Dependencies

- Projective dependencies: when the tree edges are drawn directly on a sentence, it forms a tree (without a cycle), and there is no crossing edge.
- Non-projective dependency:

Eg: A hearing is scheduled on the issue today.
Non Projective Dependencies

- which word does “on the issue” modify?
  - We scheduled a meeting on the issue today.
  - A meeting is scheduled on the issue today.

- CFGs capture only non-projective dependencies (why?)
Coordination across Constituents

- **Right-node raising:**
  - [[She bought] and [he ate]] bananas.

- **Argument-cluster coordination:**
  - I give [[you an apple] and [him a pear]].

- **Gapping:**
  - She likes sushi, and he sashimi

➡️ CFGs don’t capture coordination across constituents:
Coordination across Constituents

- She bought and he ate bananas.
- I give you an apple and him a pear.

Compare above to:

- She bought and ate bananas.
- She bought bananas and apples.
- She bought bananas and he ate apples.
The Chomsky Hierarchy

- Regular (or Right Linear) Languages
- Context-Free Languages
- Mildly Context-Sensitive Languages
- Context-Sensitive Languages
- Recursively Enumerable Languages
The Chomsky Hierarchy

<table>
<thead>
<tr>
<th>Type</th>
<th>Common Name</th>
<th>Rule Skeleton</th>
<th>Linguistic Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Turing Equivalent</td>
<td>$\alpha \rightarrow \beta$, s.t. $\alpha \neq \epsilon$</td>
<td>HPSG, LFG, Minimalism</td>
</tr>
<tr>
<td>1</td>
<td>Context Sensitive</td>
<td>$\alpha A\beta \rightarrow \alpha \gamma \beta$, s.t. $\gamma \neq \epsilon$</td>
<td>TAG, CCG</td>
</tr>
<tr>
<td>–</td>
<td>Mildly Context Sensitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Context Free</td>
<td>$A \rightarrow \gamma$</td>
<td>Phrase-Structure Grammars</td>
</tr>
<tr>
<td>3</td>
<td>Regular</td>
<td>$A \rightarrow xB$ or $A \rightarrow x$</td>
<td>Finite-State Automata</td>
</tr>
</tbody>
</table>

- Lexical Functional Grammar (LFG) (Bresnan, 1982)
- Minimalist Grammar (Stabler, 1997)

- Tree-Adjoining Grammars (TAG) (Joshi, 1969)
- Combinatory Categorial Grammars (CCG) (Steedman, 1986)
Mildly Context-Sensitive Grammar Formalisms
I. Tree Adjoining Grammar (TAG)

Some slides adapted from Julia Hockenmaier’s
TAG Lexicon (Supertags)

- Tree-Adjoining Grammars (TAG) (Joshi, 1969)
- “... super parts of speech (supertags): almost parsing” (Joshi and Srinivas 1994)
- POS tags enriched with syntactic structure
- also used in other grammar formalisms (e.g., CCG)
likes bananas with the with
Example: TAG Lexicon

α2:

NP

John

α3:

NP

tapas

α1:

S

NP

VP

VBZ

eats

NP

VP

β1:

RB

VP

VP*

always
Example: TAG Derivation

α1

α2

α3

S

NP

α1:

NP

α2:

NP

John

α3:

NP

tapas

β1:

VP

RB

always

VP*

eats
Example: TAG Derivation

\[ \alpha_1 \]
\[ \alpha_2 \quad \beta_1 \quad \alpha_3 \]

\[ S \]
\[ \text{NP} \quad \text{VP} \]
\[ \text{John} \quad \text{eats} \quad \text{tapas} \]

\[ \text{RB} \quad \beta_1 \quad \text{VP} \quad \text{VP}^* \]
\[ \text{always} \]
Example: TAG Derivation

S
  NP
  |  VP
  |   |  VP*
  |   |   |  VN
  |   |   |   |  NP
  |   |   |   |   |  tapas
  |   |   |   |  eats
  |   |   |  RB
  |   |  always
  |  John
TAG rule 1: Substitution

Derived tree:

Substitute

Derivation tree:
TAG rule 2: Adjunction

Auxiliary tree

Foot node

Derived tree:

ADJOIN

Derivation tree:
(1) Can handle long distance dependencies
(2) Cross-serial Dependencies

dat Jan Piet Marie de kinderen zag helpen laten zwemmen

- Dutch and Swiss-German
- Can this be generated from context-free grammar?
$a^n b^n$: Cross-serial dependencies

Elementary trees:

Deriving $aabb$
Tree Adjoining Grammar (TAG)

- TAG: Aravind Joshi in 1969
- Supertagging for TAG: Joshi and Srinivas 1994

- Pushing grammar down to lexicon.
- With just two rules: substitution & adjunction

- Parsing Complexity:
  - O(N^7)

- Xtag Project (TAG Penntree) (http://www.cis.upenn.edu/~xtag/)

- Local expert!
  - Fei Xia @ Linguistics (https://faculty.washington.edu/fxia/)
II. Combinatory Categorial Grammar (CCG)

Some slides adapted from Julia Hockenmaier’s
Categories

- **Categories = types**
  - Primitive categories
    - N, NP, S, etc
  - Functions
    - a combination of primitive categories
    - S/NP, (S/NP) / (S/NP), etc
    - V, VP, Adverb, PP, etc
Combinatory Rules

**Application**
- forward application:  \( \frac{x}{y} \ y \rightarrow x \)
- backward application:  \( y \ x\backslash y \rightarrow x \)

**Composition**
- forward composition:  \( \frac{x}{y} \ y/z \rightarrow x/z \)
- backward composition:  \( y\backslash z \ x\backslash y \rightarrow x\backslash z \)
- (forward crossing composition:  \( \frac{x}{y} \ y\backslash z \rightarrow x\backslash z \))
- (backward crossing composition:  \( x\backslash y \ y/z \rightarrow x/z \))

**Type-raising**
- forward type-raising:  \( x \rightarrow y \ / \ (y\backslash x) \)
- backward type-raising:  \( x \rightarrow y \ \backslash \ (y/x) \)

**Coordination \(<&>\)**
- \( x \ \text{conj} \ x \rightarrow x \)
Combinatory Rules 1 : Application

- **Forward application “>”**
  - X/Y  Y  →  X
  - (S\NP)/NP  NP  →  S\NP

- **Backward application “<“**
  - Y  X\Y  →  X
  - NP  S\NP  →  S
Function

- $\text{likes} := (S\backslash NP) / NP$
  - A transitive verb is a function from NPs into predicate $S$. That is, it accepts two NPs as arguments and results in $S$.

- Transitive verb: $(S\backslash NP) / NP$
- Intransitive verb: $S\backslash NP$
- Adverb: $(S\backslash NP) \backslash (S\backslash NP)$
- Preposition: $(NP\backslash NP) / NP$
- Preposition: $((S\backslash NP) \backslash (S\backslash NP)) / NP$
CCG Derivation:

Mary  likes  musicals
\[ NP \quad (S\backslash NP) / NP \quad NP \]

\[ S\backslash NP \quad \rightarrow \quad S \]

CFG Derivation:

Mary  likes  musicals
\[ NP \quad V \quad NP \]

\[ S \quad VP \]

\[ S \]
Combinatory Rules

- Application
  - forward application: \( x/y \ y \rightarrow x \)
  - backward application: \( y \ x\!\!\!\backslash y \rightarrow x \)

- Composition
  - forward composition: \( x/y \ y/z \rightarrow x/z \)
  - backward composition: \( y\!\!\!\backslash z \ x\!\!\!\backslash y \rightarrow x\!\!\!\backslash z \)
  - forward crossing composition: \( x/y \ y\!\!\!\backslash z \rightarrow x\!\!\!\backslash z \)
  - backward crossing composition: \( x\!\!\!\backslash y \ y/z \rightarrow x/z \)

- Type-raising
  - forward type-raising: \( x \rightarrow y / (y\!\!\!\backslash x) \)
  - backward type-raising: \( x \rightarrow y \!\!\!\backslash (y/x) \)

- Coordination \(<\&>\)
  - \( x \ conj \ x \rightarrow x \)
Combinatory Rules 4: Coordination

- $X \text{ conj } X \Rightarrow X$

- Alternatively, we can express coordination by defining conjunctions as functions as follows:

  - $\text{and} := (X \backslash X) / X$
Coordination with CCG

Examples from Prof. Mark Steedman
Coordination with CCG

- Marcel
- conjectured
- and
- proved
- completeness

NP

(S\NP)/NP

(X\X)/X

(S\NP)/NP

NP

Application

- forward application: x/y y → x
- backward application: y x\y → x
Coordination with CCG

Marcel
NP
\frac{conjectured}{(S\backslash NP)/NP}
\frac{and}{(X\backslash X)/X}
\frac{proved}{(S\backslash NP)/NP}
\frac{completeness}{NP}
\frac{((S\backslash NP)/NP)\backslash((S\backslash NP)/NP)}{<}
\frac{(S\backslash NP)/NP}{<}
\frac{(S\backslash NP)/NP}{<}
\frac{(S\backslash NP)/NP}{>}
\frac{S\backslash NP}{>}
\frac{S}{<}

- Application
- forward application: x/y y \rightarrow x
- backward application: y x\backslash y \rightarrow x
Combinatory Rules

- **Application**
  - forward application: \( x/y \ y \rightarrow x \)
  - backward application: \( y \ x/y \rightarrow x \)

- **Composition**
  - forward composition: \( x/y \ y/z \rightarrow x/z \)
  - backward composition: \( y/z \ x/y \rightarrow x/z \)
  - forward crossing composition: \( x/y \ y/z \rightarrow x/z \)
  - backward crossing composition: \( x/y \ y/z \rightarrow x/z \)

- **Type-raising**
  - forward type-raising: \( x \rightarrow y / (y\x) \)
  - backward type-raising: \( x \rightarrow y \ (y/x) \)

- **Coordination \(<&>\)**
  - \( x \ conj \ x \rightarrow x \)
### Coordination with CCG

<table>
<thead>
<tr>
<th>Marcel</th>
<th>NP</th>
<th>conjectured</th>
<th>(S\NP)/NP</th>
<th>and</th>
<th>(X\X)/X</th>
<th>might</th>
<th>(S\NP)/(S\NP)</th>
<th>prove</th>
<th>(S\NP)/NP</th>
<th>completeness</th>
</tr>
</thead>
</table>

**Application**
- **forward application:** \(x/y ~ y \rightarrow x\)
- **backward application:** \(y ~ x/y \rightarrow x\)

**Composition**
- **forward composition:** \(x/y ~ y/z \rightarrow x/z\)
- **backward composition:** \(y/z ~ x/y \rightarrow x\)
- **forward crossing composition:** \(x/y ~ y/z \rightarrow x\)
- **backward crossing composition:** \(x/y ~ y/z \rightarrow x\)

- **Example**

  - **Forward Application:** Marcel conjectured that and might prove completeness.
  - **Composition:** Marcel conjectured that and might prove completeness.
Coordination with CCG

- **Application**
  - forward application: \( x/y \ y \rightarrow x \)
  - backward application: \( y \ x/y \rightarrow x \)

- **Composition**
  - forward composition: \( x/y \ y/z \rightarrow x/z \)
  - backward composition: \( y/z \ x/y \rightarrow x/z \)
  - forward crossing composition: \( x/y \ y/z \rightarrow x/z \)
  - backward crossing composition: \( x/y \ y/z \rightarrow x/z \)
Combinatory Rules

- **Application**
  - forward application: \( x/y \ y \rightarrow x \)
  - backward application: \( y \ x\backslash y \rightarrow x \)

- **Composition**
  - forward composition: \( x/y \ y/z \rightarrow x/z \)
  - backward composition: \( y\backslash z \ x\backslash y \rightarrow x\backslash z \)
  - forward crossing composition: \( x/y \ y\backslash z \rightarrow x\backslash z \)
  - backward crossing composition: \( x\backslash y \ y/z \rightarrow x/z \)

- **Type-raising**
  - forward type-raising: \( x \rightarrow y / (y\backslash x) \)
  - backward type-raising: \( x \rightarrow y \backslash (y/x) \)

- **Coordination \(<\&>\)**
  - \( x \text{ conj} x \rightarrow x \)
Combinatory Rules 3 : Type-Raising

- Turns an argument into a function

- Forward type-raising: \( X \rightarrow T / (T\setminus X) \)
- Backward type-raising: \( X \rightarrow T \setminus (T/X) \)

For instance...

- Subject type-raising: \( NP \rightarrow S / (S \setminus NP) \)
- Object type-raising: \( NP \rightarrow (S\setminus NP) \setminus ((S\setminus NP) / NP) \)
Combinatory Rules 3 : Type-Raising

<table>
<thead>
<tr>
<th></th>
<th>dislike</th>
<th>and</th>
<th>Mary</th>
<th>likes</th>
<th>musicals</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>NP</td>
<td>(S\NP)/NP</td>
<td>CONJ</td>
<td>NP</td>
<td>(S\NP)/NP</td>
</tr>
</tbody>
</table>

- **Application**
  - forward application: \( x/y \ y \rightarrow x \)
  - backward application: \( y \ x/y \rightarrow x \)

- **Type-raising**
  - forward type-raising: \( x \rightarrow y / (y\times) \)
  - backward type-raising: \( x \rightarrow y \times (y\times) \)
  - Subject type-raising: \( \text{NP} \rightarrow S / (S \setminus \text{NP}) \)
  - Object type-raising: \( \text{NP} \rightarrow (S\setminus\text{NP}) \setminus ((S\setminus\text{NP}) / \text{NP}) \)

- **Coordination <&>**
  - \( x \text{ conj } x \rightarrow x \)
Combinatory Rules 3: Type-Raising

\[
\begin{align*}
S/(S\backslash NP) & \xrightarrow{T} S/((S\backslash NP)/NP) \\
& \xrightarrow{B} S/NP \\
& \xrightarrow{B} S
\end{align*}
\]
Combinatory Categorial Grammar (CCG)

- CCG: Steedman in 1986

- Pushing grammar down to lexicon.
- With just a few rules: application, composition, type-raising

- We’ve looked at only syntactic part of CCG
- A lot more in the semantic part of CCG (using lambda calculus)

- Parsing Complexity:
  - O(N^6)

- Local expert!