Natural Language Processing 1

MSc Artificial Intelligence

Lecturer: Tejaswini Deoskar
Institute for Logic, Language, and Computation

Fall 2016, week 3, lecture b

some slides adapted from F. Keller and S. Clark.

Today:

- Dependency Grammars and Dependency Parsing
- Introduction to Combinatory Categorial Grammar (CCG) and parsers

Constituents vs. Dependencies

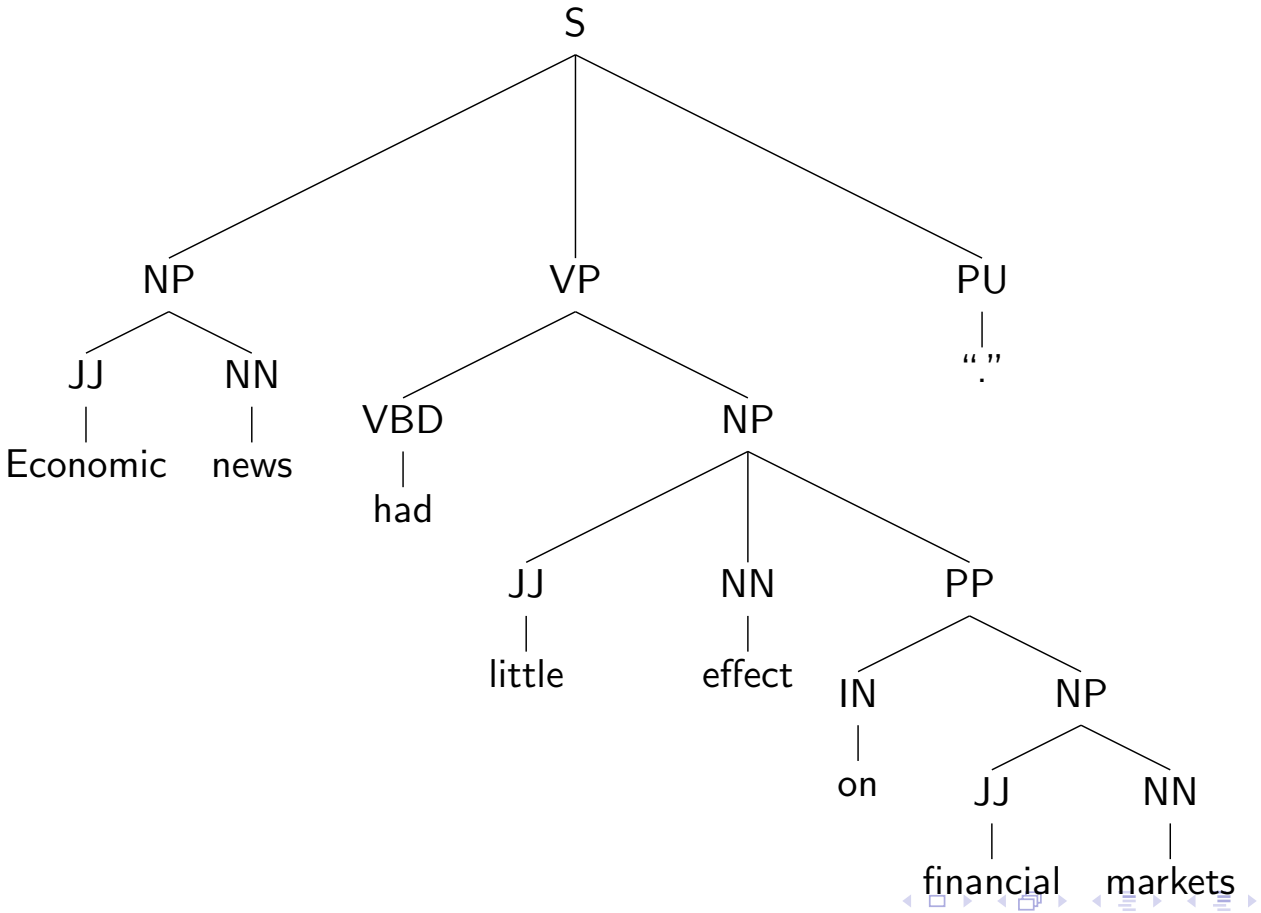
Phrase-structure grammars model constituent structure : the configurational patterns in a language.

For e.g., verb phrases have certain properties in English

- *I like bananas. Do you?* [VP-ellipses]
- *I [like bananas] but [hate pineapples].* [VP- co-ordination]
- *I said I would meet Fred, and [meet Fred] I did.* [VP- fronting]

In other languages (like e.g., German), there is less evidence of the existence of a VP.

Constituency Tree



Constituents vs. Dependencies

From a semantic point of view, the important thing about verbs such as *like* is that they license two NPs

- an agent, found in subject position (the actor, taking Nominative case)
- a patient, found in object positions (the entity acted upon, taking Accusative case).

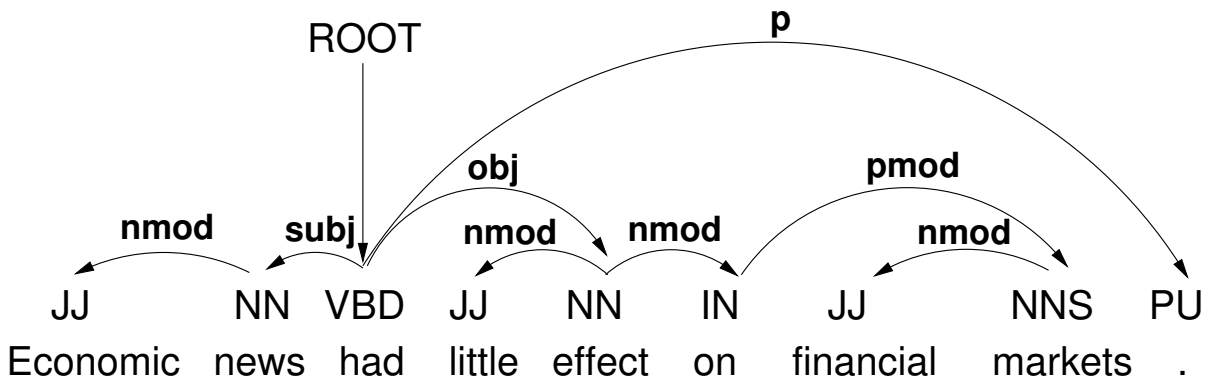
Which arguments are licensed, and what roles they play depends on the verb (configurational is secondary)

To account for semantic patterns, we focus on dependencies directly. Dependencies can be easily identified even in non-configurational (free-word-order) languages.

Dependency Structure

A dependency structure consists of dependency relations, which are binary and asymmetric. A relation consists of

- a head (H)
- a dependent (D)
- a label identifying the relation between H and D



[From Joakim Nivre, Dependency Grammar and Dependency Parsing.]

Dependency Trees

Formally, the dependency structure of a sentence is a graph with the words of the sentence as its nodes, linked by directed, labeled edges, with the following properties:

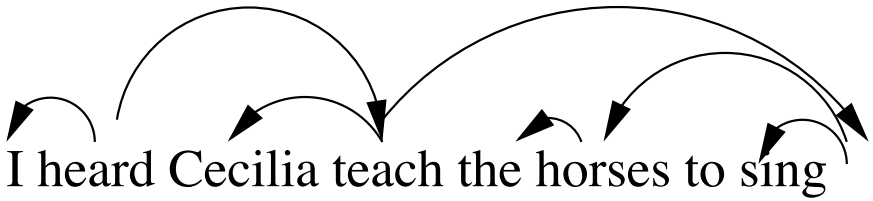
- connected: every node is related to at least one other node, and (through transitivity) to ROOT
- single headed: every node (except ROOT) has exactly one incoming edge (from its head)
- acyclic: the graph cannot contain cycles of directed edges.

These conditions ensure that the dependency structure is a tree.

Dependency Trees: Projectivity

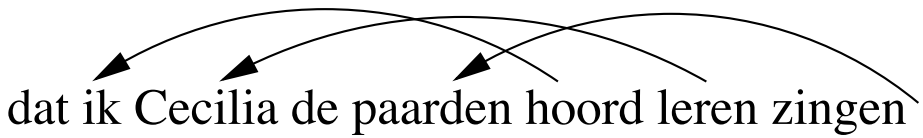
We distinguish projective and non-projective dependency trees:

A dependency tree is projective wrt. a particular linear order of its nodes if, for all edges $h \rightarrow d$ and nodes w, w occurs between h and d in linear order only if w is dominated by h.



Dependency Trees: Non-projectivity

A dependency tree is non-projective if w can occur between h and d in linear order without being dominated by h.



that I Cecilia the horses heard teach sing

"That I heard Cecilia teach the horses to sing"

A non-projective dependency grammar is not context-free. But efficient non-projective parsers exist, e.g., the MST parser.

Constituent versus Dependency parsing

Why consider dependency parsing as a distinct topic?

- context-free parsing algorithms base their decisions on adjacency
- in a dependency structure, a dependent need not be adjacent to its head (even if the structure is projective);
- we need new parsing algorithms to deal with non-adjacency (and with non-projectivity if present).

Two types of Dependency Parsers

- graph-based dependency parsing, based on maximum spanning trees (MST parser, McDonald et al. 2005)
- transition-based dependency parsing, based on shift-reduce parsing (MALT parser, Nivre 2003)

Alternative: map dependency trees to phrase structure trees and do standard CF parsing *(not covered here)*.

Graph-based dependency Parsing

Goal: find the highest scoring dependency tree in the space of all possible trees for a sentence.

Let $x = x_1 \ldots x_n$ be the input sentence, and y a dependency tree for x.
Here, y is a set of dependency edges, with $(i, j) \in y$ if there is an edge from x_i to x_j .

In order to assign scores to dependency trees, we edge factorize them: the score of a tree is the sum of the scores of all its edges.

Graph-based dependency Parsing

The score of a dependency edge (i,j) is:

$$s(i,j) = \mathbf{w} \cdot \mathbf{f}(i,j)$$

where $\mathbf{w}$ is a weight vector and $\mathbf{f}(i,j)$ is a feature vector. Then the score of dependency tree y for a sentence x is

$$s(x,y) = \sum_{(i,j) \in y} s(i,j) = \sum_{(i,j) \in y} \mathbf{w} \cdot \mathbf{f}(i,j)$$

Dependency parsing is the task of finding the tree y with highest score for a given sentence x.

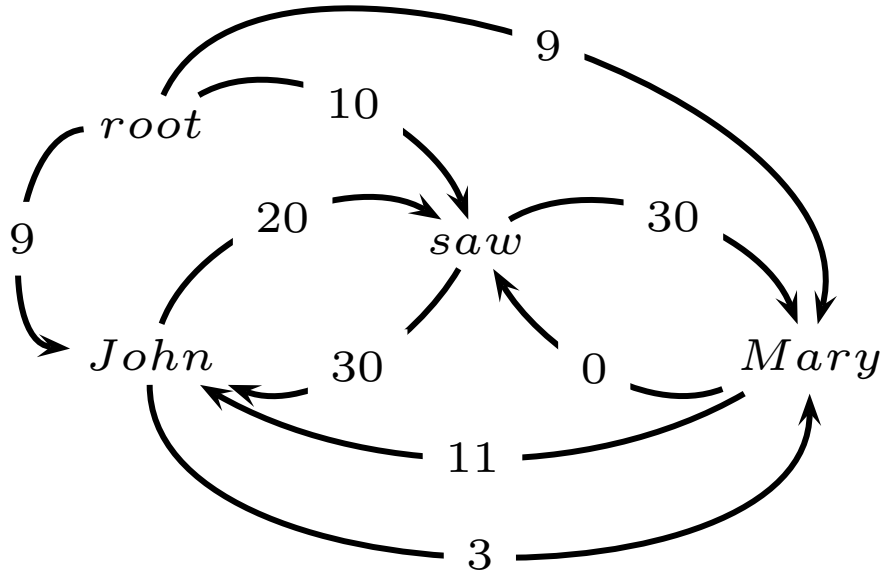
Maximum Spanning Tree parsing

This task can be achieved using the following approach (McDonald et al. 2005):

- start with a totally connected graph G, i.e., assume a directed edge between every pair of words
- assume you have a scoring function that assigns a score $s(i, j)$ to every edge (i, j)
- find the maximum spanning tree (MST) of G, i.e., the tree with the highest overall score that includes all nodes of G
- this is possible in $O(n^2)$ time using the Chu-Liu-Edmonds algorithm; it finds both projective and non-projective MSTs
- the correct parse is the MST of G

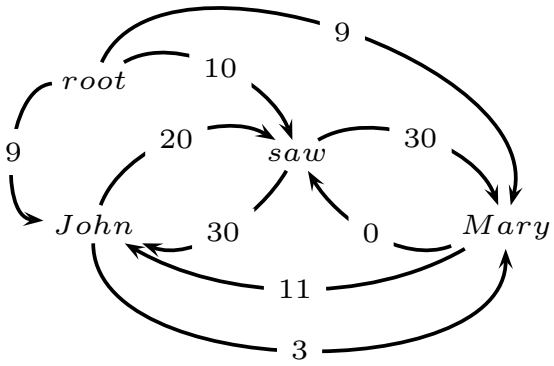
Chu-Liu-Edmonds (CLE) Algorithm

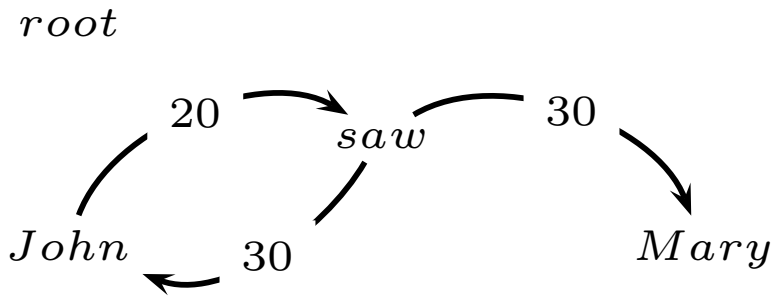
Example : **x** $=$ *John saw Mary*, with Graph G_x. Start with the fully-connected graph, with scores:



Chu-Liu-Edmonds (CLE) Algorithm

Each node j in the graph greedily selects the incoming edge the highest score $s(i, j)$:

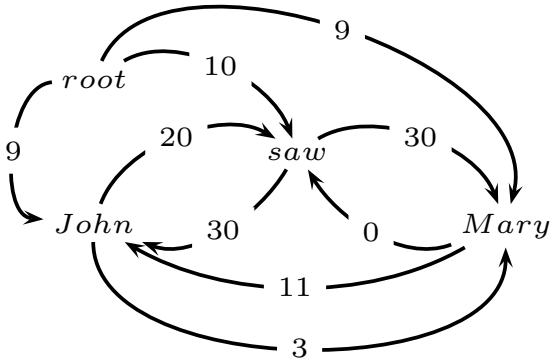


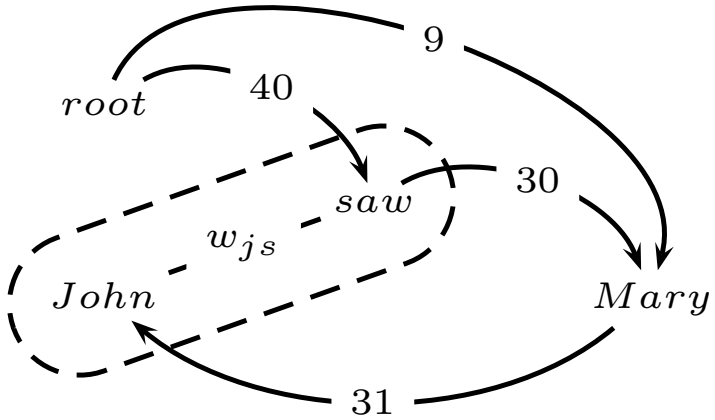


If a tree results, it must be the MST. If not, there must be a cycle.

CLE Algorithm: Recursion

Identify the cycle and contract it into a single node and recalculate scores of incoming and outgoing edges

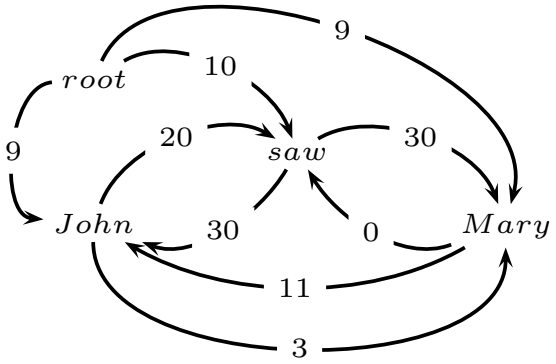


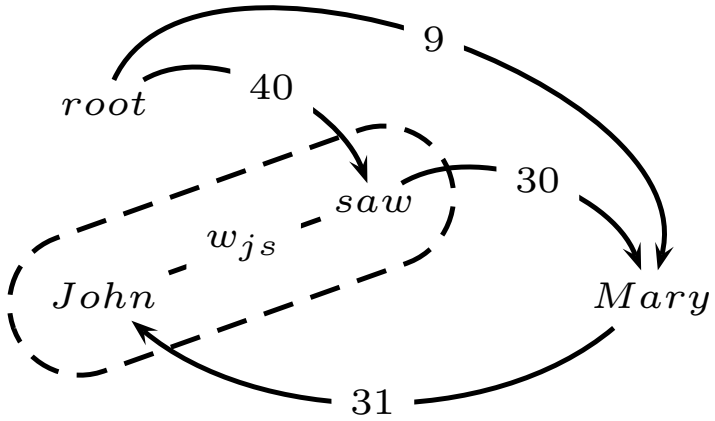


- Incoming arc weights: Equal to the weight of best spanning tree that includes head of incoming arc, and all nodes in cycle
  - $Root \rightarrow saw \rightarrow John$ is 40; $Root \rightarrow John \rightarrow saw$ is 29; hence select $Root \rightarrow saw$
- Outgoing arc weights: Equal to the max of outgoing arcs over all vertices in the cycle
  - $John \rightarrow Mary$ is 3, and $saw \rightarrow Mary$ is 30; hence select $saw \rightarrow Mary$

CLE Algorithm: Recursion

Identify the cycle and contract it into a single node and recalculate scores of incoming and outgoing edges

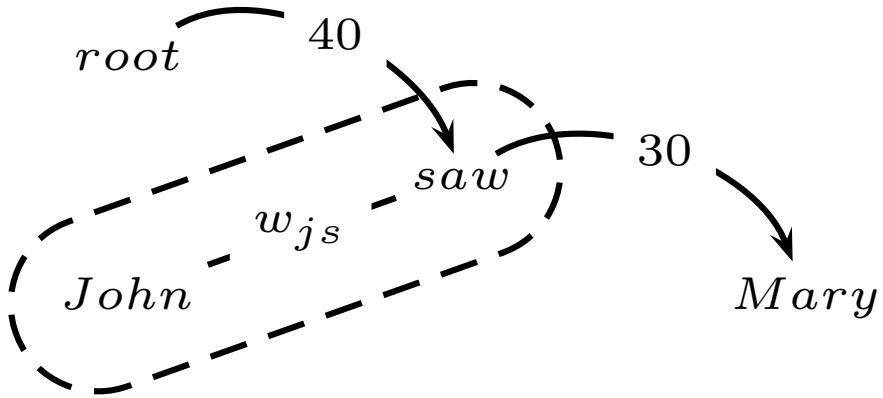




- Now call CLE recursively on this contracted graph.
- MST on the contracted graph is equivalent to MST on the original graph.

CLE Algorithm: Recursion

- Again, greedily collect incoming edges to all nodes

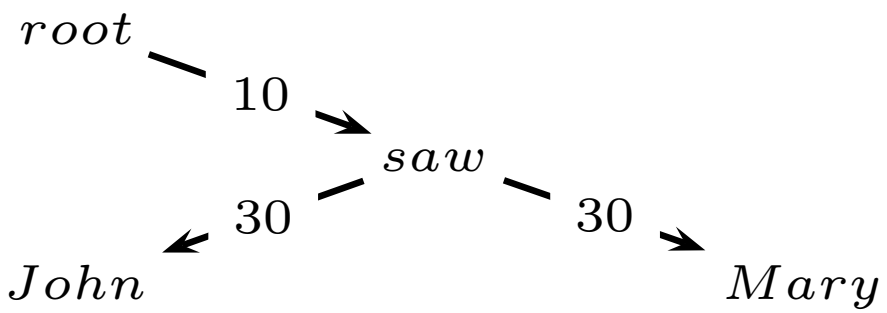


- This is a tree, hence it must be the MST of the graph.

CLE Algorithm: Reconstruction

Now reconstruct the uncontracted graph: MST Parser:

- the edge from w_{js} to *Mary* was from *saw*.
- The edge from ROOT to w_{js} was a tree from $ROOT$ to *saw* to *John*, so we include these edges too



MST Parser: Features

- Parsing accuracy depends the scoring function, i.e., the features $\mathbf{f}(i, j)$ and the weight vector $\mathbf{w}$.

Features $\mathbf{f}(i, j)$ used:

Unigram Features
p-word, p-pos
p-word
p-pos
c-word, c-pos
c-word
c-pos

Bigram Features
p-word, p-pos, c-word, c-pos
p-pos, c-word, c-pos
p-word, c-word, c-pos
p-word, p-pos, c-pos
p-word, p-pos, c-word
p-word, c-word
p-pos, c-pos

p: parent; c: child

MST Parser: More Features

More features:

In Between PoS Features
p-pos, b-pos, c-pos
Surrounding Word PoS Features
p-pos, p-pos+1, c-pos−1, c-pos
p-pos−1, p-pos, c-pos−1, c-pos
p-pos, p-pos+1, c-pos, c-pos+1
p-pos−1, p-pos, c-pos, c-pos+1

p: parent, c: child, b: between parent and child, −1: to left, +1: to right.

Total feature: 6,998,447 for English 13,450,672 for Czech.

Transition-based Dependency Parsing

Main alternative is MALT parser (Nivre, 2003)

- the MALT parser is a shift-reduce parser, defined through a transition system
- for a given parse state, the transition system defines a set of actions T which the parser can take
- if more than one action is applicable, a classifier (e.g., an SVM) is used to decide which action to take
- just like in the MST model, this requires a large set of hand-engineered features

Basic Shift-Reduce Parsing

- Shift-reduce parsing is a depth-first, and bottom-up strategy
- Basic shift-reduce parser repeatedly:
  - Shifts input symbols onto a stack
  - Whenever possible, reduces one or more items from top of stack that match RHS of rule, replacing with LHS of rule.
- Needs to maintain backtrack points.

Shift-reduce example

- Grammar

$$
\begin{array}{lcl}
S & \rightarrow & NP\ VP \\
NP & \rightarrow & DT\ NN \\
VP & \rightarrow & V\ NP \\
DT & \rightarrow & the \\
NN & \rightarrow & dog|bit \\
V & \rightarrow & dog|bit
\end{array}
$$

- Operations
  - Shift (S)
  - Reduce (R)
  - Backtrack to step n (Bn)

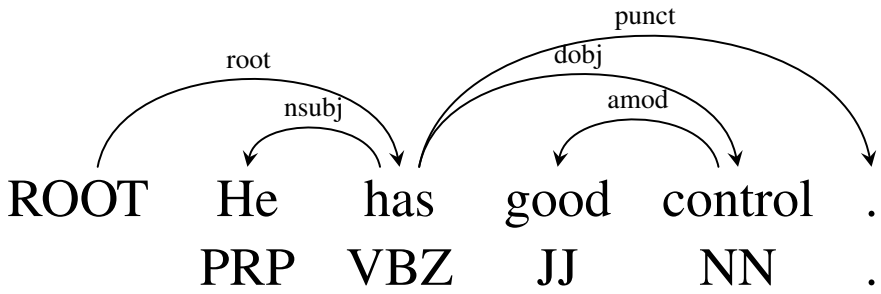
Step	Op.	Stack	Input
0			the dog bit
1	S	the	dog bit
2	R	DT	dog bit
3	S	DT dog	bit
4	R	DT V	bit
5	S	DT V bit	
6	R	DT V V	
7	B5	DT V bit	
8	R	DT V NN	
9	B3	DT dog	bit
10	R	DT NN	bit
11	R	NP	bit
12	R	NP bit	bit
...			

Transition-based Dependency Parsing

The arc-standard transition system (Nivre 2003)

- configuration $c = (s, b, A)$ with stack s, buffer b, set of dependency arcs A
- initial configuration for sentence $w_1 \ldots w_n$ is $s = [ROOT]$, $b = [w_1, \ldots, w_n]$, $A = \phi$
- c is terminal if buffer is empty, stack contains only $ROOT$, and parse tree is given by A_c
- if s_i is the i^{th} top element on stack, and b_i the i^{th} element on buffer, then we have the following transitions:
  - LEFT-ARC(l): adds arc $s_1 \rightarrow s_2$ with label l and removes s_2 from stack; precondition: $|s| >= 2$
  - RIGHT-ARC(l): adds arc $s_2 \rightarrow s_1$ with label l and removes s_1 from stack; precondition: $|s| >= 2$
  - SHIFT: moves b_1 from buffer to stack; recondition: $|b| >= 1$

Transition-based Dependency Parsing



Transition	Stack	Buffer	A
	[ROOT]	[He has good control .]	$\emptyset$
SHIFT	[ROOT He]	[has good control .]	
SHIFT	[ROOT He has]	[good control .]	
LEFT-ARC(nsubj)	[ROOT has]	[good control .]	$A \cup$ nsubj(has,He)
SHIFT	[ROOT has good]	[control .]	
SHIFT	[ROOT has good control]	[.]	
LEFT-ARC(amod)	[ROOT has control]	[.]	$A \cup$amod(control,good)
RIGHT-ARC(dobj)	[ROOT has]	[.]	$A \cup$ dobj(has,control)
...	...	...	...
RIGHT-ARC(root)	[ROOT]	[]	$A \cup$ root(ROOT,has)

Comparing the MST and MALT parsers

- the MST parser selects the globally optimal tree, given a set of edges with scores
- it can naturally handle projective and non-projective trees
- the MALT parser makes a sequence of local decisions about the best parse action
- it can be extended to "pseudo-projective" dependency trees with transformation techniques
- accuracy of MST and MALT is similar, but MALT is faster
- both require a set of manually engineered features and a model that learns feature weights

Recent work on dependency parsing uses neural networks to learn both features and feature weights

Traditional Dependency Parsing

- classifier chooses which transition (parser action) to take for each word in the input sentence
- features for classifier similar to MALT parser
  - word/PoS unigrams, bigrams, trigrams
  - state of the parser
  - dependency tree build so far
- Problem:
  - feature templates need to be hand-crafted
  - millions of features
  - features are sparse and slow to extract

NN based Dependency Parsing

Chen and Manning (2004) propose:

- keep the simple shift-reduce parser
- replace the classifier for transitions with a neural net
- use dense features (embeddings) instead of sparse, handcrafted features
- Results:
  - efficient parser (up to twice as fast as standard MALT parser)
  - good performance (about 2% higher precisions than MALT)

Combinatory Categorial Grammar (CCG)

- Categorial grammar CG is one of the oldest grammar formalisms (Ajdukiewicz, 1935; Bar-Hillel, 1953; Lambek 1958)
- Various flavours of CG now available: type-logical CG, algebraic pre-group grammars (Lambek), CCG
- CCG is now an established linguistic formalism (Steedman, 1996, 2000)
  - syntax; semantics; prosody; information structure; wide-coverage parsing; incremental parsing, generation, etc.

CCG slides from Stephen Clark

Combinatory Categorial Grammar (CCG)

- CCG is a strongly lexicalised grammar
- An elementary syntactic structure - a lexical category - is assigned to each word in a sentence.

  walked : $S \setminus NP$ *(give me an NP to my left and I return a sentence)*

- A semantic interpretation is also assigned to each word

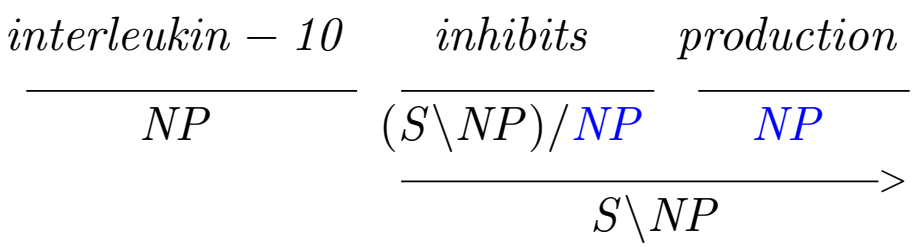
  walked : $\lambda x.WALK(x)$

- A small number of rules define how categories can combined - rules are called combinators

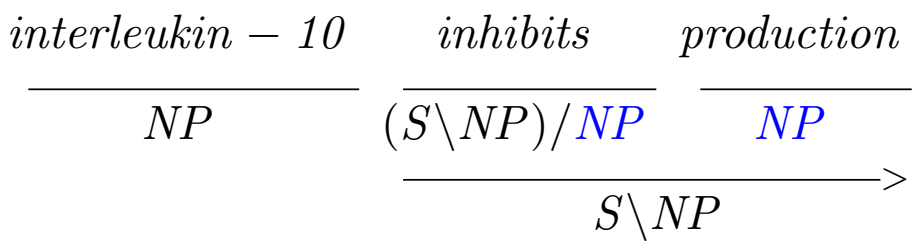
CCG Lexical Categories

- Atomic categories : S, N, NP, PP (not many more)
- Complex categories are built recursively from atomic categories and slashes, which indicate the directions of arguments
- Complex categories encode 'subcategorisation' information
  - intransitive verb: $S \setminus NP$: *walked*
  - transitive verb: $(S \setminus NP)/NP$: *respected*
  - ditransitive verb : $((S \setminus NP)/NP)/NP$: *gave*
- Complex categories can encode modification
  - PP nominal : $(NP \setminus NP)/NP$
  - PP verbal : $((S \setminus NP) \setminus (S \setminus NP))/NP$

A Simple CCG Derivation

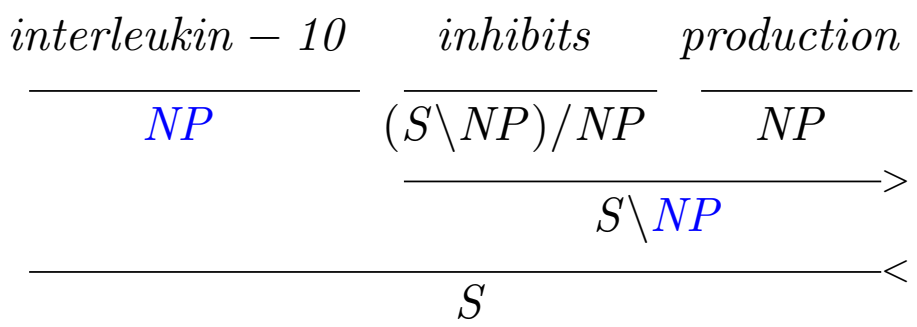


A Simple CCG Derivation



> forward application

A Simple CCG Derivation



$>$ forward application

$<$ backward application

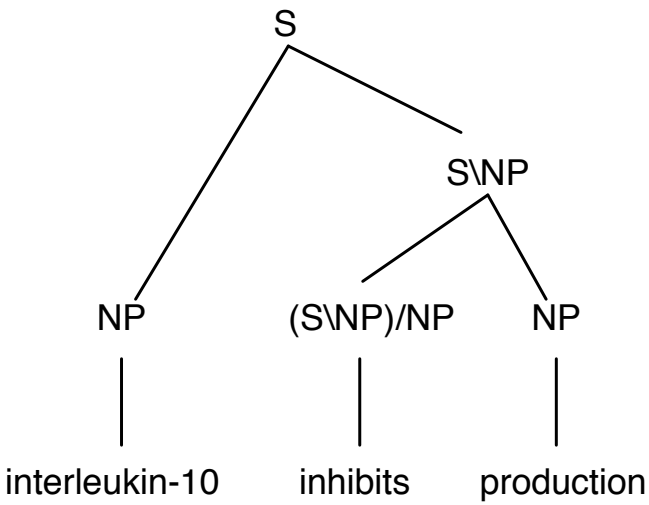
Function Application Rule Schemata

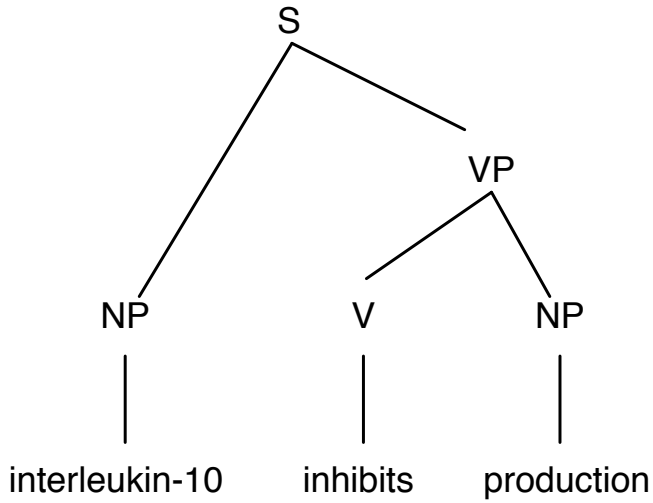
- Forward ($>$) and backward ($<$) application:

$$X/Y \quad Y \quad \Rightarrow \quad X \quad (>)$$
$$Y \quad X\backslash Y \quad \Rightarrow \quad X \quad (<)$$

Classical Categorial Grammar

- Only has application
- Is context-free





Extraction out of a Relative Clause

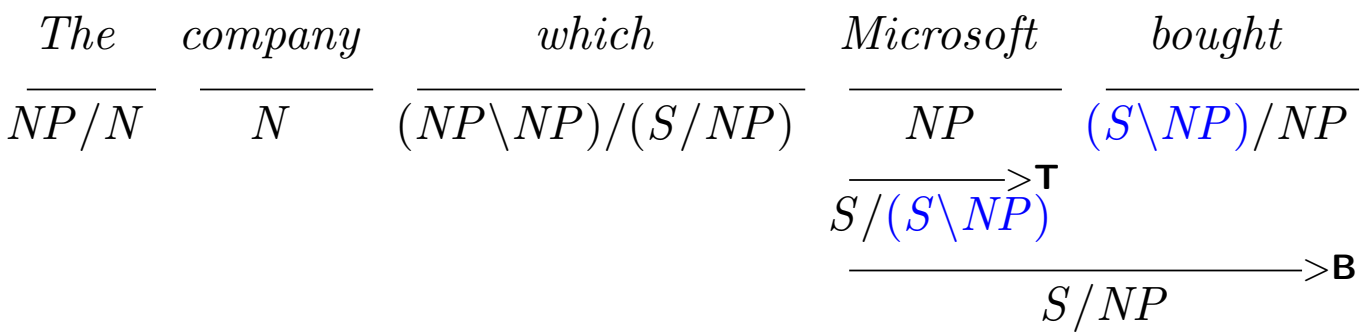
$$\frac{\textit{The}}{NP/N} \quad \frac{\textit{company}}{N} \quad \frac{\textit{which}}{(NP\backslash NP)/(S/NP)} \quad \frac{\textit{Microsoft}}{NP} \quad \frac{\textit{bought}}{(S\backslash NP)/NP}$$

Extraction out of a Relative Clause

$$\frac{The}{NP/N} \quad \frac{company}{N} \quad \frac{which}{(NP\backslash NP)/(S/NP)} \quad \cfrac{\frac{Microsoft}{NP}}{S/(S\backslash NP)}{\scriptstyle >\mathsf{T}} \quad \frac{bought}{(S\backslash NP)/NP}$$

$>$ Type-raising

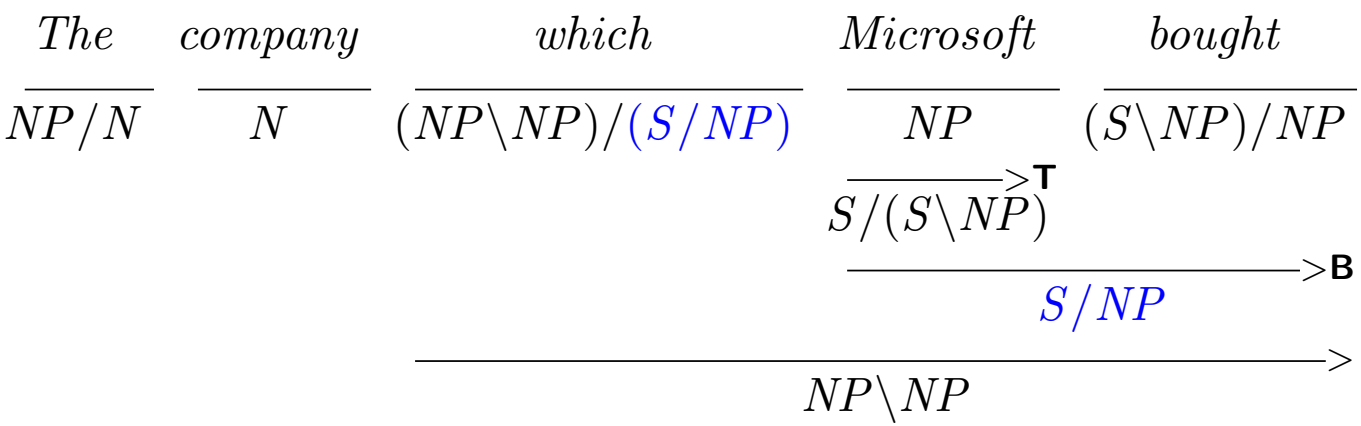
Extraction out of a Relative Clause



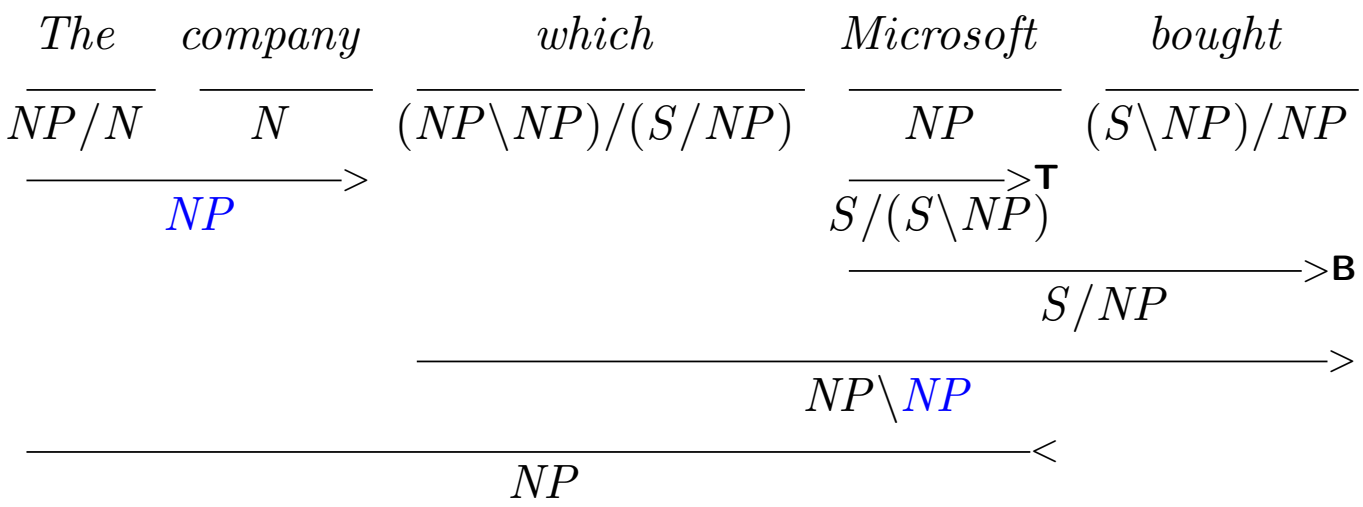
$>$ **T** Type-raising

$>$ **B** forward composition

Extraction out of a Relative Clause



Extraction out of a Relative Clause



Forward Composition and Type-Raising

- Forward composition ($>_B$):

$$X/Y \quad Y/Z \Rightarrow X/Z \quad (>_B)$$

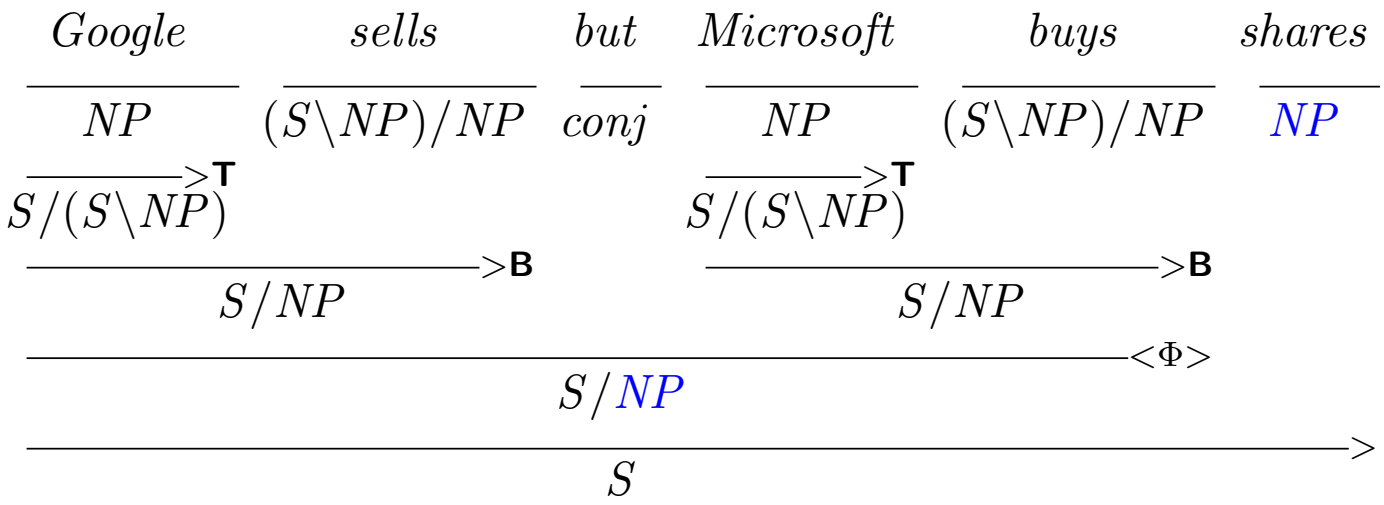
- Type-raising ($\mathbf{T}$):

$$X \Rightarrow T/(T \setminus X) \quad (>_T)$$

$$X \Rightarrow T \setminus (T/X) \quad (<_T)$$

- Extra combinatory rules increase the weak generative power to mild context-sensitivity

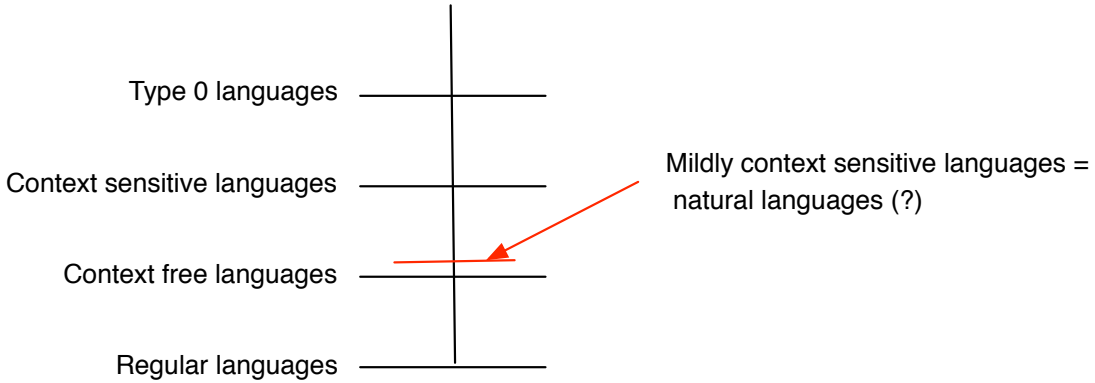
Non-constituents in CCG (Right Node Raising)



- $>$ T type-raising
- $>$ B forward composition

CCG

- CCG is mildly context-sensitive
- Natural language is provably non-context-free
- Construction in Dutch and Swiss German (Shieber, 1985) require more than context-free power for their analysis
  - these have *crossing* dependencies, which CCG can handle.



Wide-coverage CCG parsing

- CCGBank developed by Hockenmaier and Steedman (Hockenmaier & Steedman, 2003)
- Phrase-structure tree in Penn Treebank (semi)-automatically converted into CCG derivations.

Inducing a grammar

- Grammar (lexicon) read off the leaves of the tree
  - 1200 lexical category types in CCGBank (compare to 45 POS tags in PTB)
- In addition to the grammar, CCGBank provides training data for the statistical model.

Parsing with CCG

- Stage 1 :
  - Assign POS tags and lexical categories to words in the sentence
  - Use taggers to assign POS tags and categories (based on standard tagging techniques)
- Stage 2:
  - Combine the categories using the combinatory rules
  - Can use standard bottom-up CKY chart-parsing algorithms
- Stage 3:
  - Find the highest scoring derivation according to some model (generative, CRF, perceptron)
  - Viterbi algorithm finds this efficiently.

Why CCG

- Formally, generative capacity matches human languages: mild-context sensitivity
- Transparent (one-to-one) mapping between syntax and semantics
- Currently emerging as dominant framework for deep semantic tasks: Question-answering, Textual Entailment.

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-

-
-