Statistical Natural Language Processing

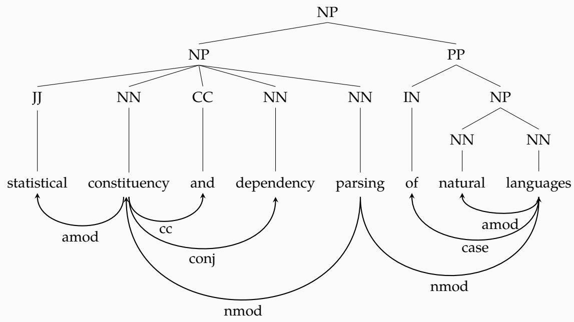
Statistical Parsing

Çağrı Çöltekin

University of Tübingen
Seminar für Sprachwissenschaft

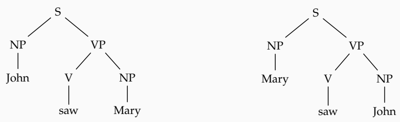
Summer Semester 2018

Next few lectures are about



Why do we need syntactic parsing?

- Syntactic analysis is an intermediate step in (semantic) interpretation of sentences



  As result, it is useful for applications like *question answering*, *information extraction*, …
- (Statistical) parsers are also used as *language models* for applications like *speech recognition* and *machine translation*
- It can be used for *grammar checking*, and can be a useful tool for linguistic research

Ingredients of a parser

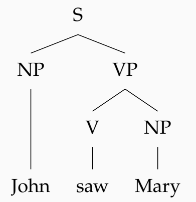
- A grammar
- An algorithm for parsing
- A method for ambiguity resolution

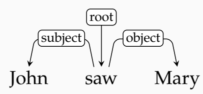
Formal grammars

- A formal grammar is a finite specification of a (possibly infinite) language
- We are interested in two broad classes of grammars
  Constituency (or phrase structure) grammars
  Dependency grammars
- Various theories of 'grammar' (e.g., HPSG, LFG, CCG) use ideas/notions from both
- We will study these grammars in their relation to parsing, we do not study or focus on any specific theory

Dependency vs. constituency

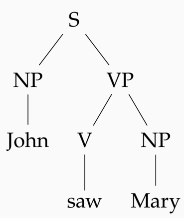
- Constituency grammars are based on units formed by a group of lexical items (constituents or phrases)
- Dependency grammars model binary head–dependent relations between words
- Most of the theory of parsing is developed with constituency grammars
- Dependency grammars has recently become popular in CL





Constituency grammars

- Constituency grammars are probably the most studied grammars both in linguistics, and computer science
- The main idea is that groups of words form natural groups, or 'constituents', like *noun phrases* or *word phrases*
- *phrase structure grammars* or *context-free grammars* are often used as synonyms



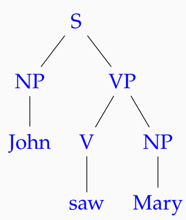
Note: many grammar formalisms posit a particular form of constituency grammars, we will not focus on a particular grammar formalism here.

Formal definition

A phrase structure grammar is a tuple (Σ, N, S, R)

Σ is a set of terminal symbols
N is a set of non-terminal symbols
$S \in N$ is a distinguished *start* symbol
R is a set of 'rewrite' rules of the form $\alpha A \beta \rightarrow \gamma$ for $A \in N$ $\alpha, \beta, \gamma \in \Sigma \cup N$

- The grammar accepts a sentence if it can be derived from S with the rewrite rules R



$$\begin{array}{lll} S & \rightarrow & NP\ VP \\ NP & \rightarrow & John \mid Mary \end{array} \qquad \begin{array}{lll} VP & \rightarrow & V\ NP \\ V & \rightarrow & saw \end{array}$$

Example derivation

The example grammar:

S	$\rightarrow$	NP VP	VP	$\rightarrow$	V NP
NP	$\rightarrow$	John | Mary	V	$\rightarrow$	saw

- Phrase structure grammars derive a sentence with successive application of rewrite rules.
  S $\Rightarrow$NP VP $\Rightarrow$John VP $\Rightarrow$John V NP $\Rightarrow$John saw NP $\Rightarrow$John saw Mary
  or, S $\stackrel{*}{\Rightarrow}$John saw Mary
- The intermediate forms that contain non-terminals are called *sentential forms*

Chomsky hierarchy of grammars

type 0 Recursively enumerable, recognized by Turing machines (HPSG, LFG)
$\alpha A \beta \rightarrow \gamma$

type 1 Context sensitive, recognized by linear-bound automaton
$\alpha A \beta \rightarrow \alpha \gamma \beta, \quad \gamma \neq \epsilon$

type 2.1 Mildly context sensitive (TAG, CCG)

type 2 Context free, recognized by push-down automata
$A \rightarrow \alpha$

type 3 Regular, recognized by finite-state automata
$A \rightarrow aB$ or $A \rightarrow Ba$

In all of the above A and B are non-terminals, a is a terminal symbol, α, β, γ are sequences of terminals and non-terminals, and ϵ is the empty string.

Some examples

- Regular grammars (finite-state automata) do not have any memory
  - can represent a^*b^*, but not $a^n b^n$
- Finite-state automata are used in many tasks in CL, including morphological analysis, partial parsing
- Context free grammars (push-down automata) uses a stack
  - can represent $a^n b^n$, $a^n b^m c^m d^n$, but not $a^n b^m c^n d^m$
- Context-free grammars form the basis of most parsers
- Context-sensitive languages can do all of the above, but they are too powerful, and computationally expensive

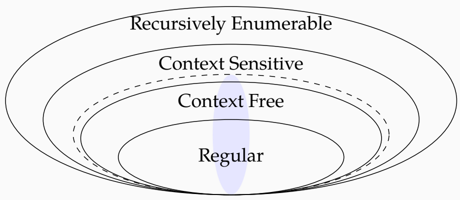
Expressiveness of grammar classes

- The class of grammars adequate for formally describing natural languages has been an important question for (computational) linguistics
- For the most part, context-free grammars are enough, but there are some examples, e.g., from Swiss German (Shieber 1985) Jan säit das…

…mer em Hans es huss hälfed aastriiche
…we Hans (DAT) the house (ACC) helped paint

Note that this resembles $a^n b^m c^n d^m$.

Chomsky hierarchy: the picture



- Chomsky hierarchy of languages form a hierarchy (with some care about empty language)
- It is often claimed that mildly context sensitive grammars (dashed ellipse) are adequate for representing natural languages
- Note, however, not even every regular language is a potential natural language (e.g., a^*bbc^*). The possible natural languages probably cross-cut this hierarchy (shaded region)

Constituency grammars and parsing

- Context-free grammars are parseable in $O(n^3)$ time complexity using dynamic programming algorithms
- Mildly context-sensitive grammars can also be parsed in polynomial time ($O(n^6)$)
- Polynomial time algorithms are not always good enough in practice
  - We often use approximate solutions with greedy search algorithms

Where do grammars come from

- Grammars for (statistical) parsing can be either
  - hand crafted (many years of expert effort)
  - extracted from *treebanks* (which also require lots of effort)
  - 'induced' from raw data (interesting, but not as successful)
- Current practice relies mostly on treebanks
- Hybrid approaches also exist
- Grammar induction is not common (for practical models) but exploiting unlabled data is also a common trend

Grammars for natural language parsing: summary

- A grammar is a formal device for specifying a language
- Grammars are one of the important components of a parser, they can be hand-crafted or extracted from a treebank
- Most of the parsing theory and practice is based on constituency, particularly context-free, grammars
- Dependency grammars have become more popular recently, and often easier to use in NLP applications

Context free grammars
recap

- Context free grammars are sufficient for expressing most phenomena in natural language syntax
- Most of the parsing theory (and practice) is build on parsing CF languages
- The context-free rules have the form

$$A \rightarrow \alpha$$

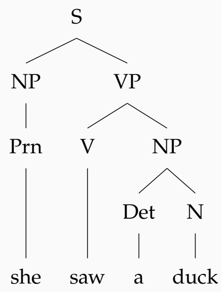
where A is a single non-terminal symbol and α is a (possibly empty) sequence of terminal or non-terminal symbols

An example context-free grammar

S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the

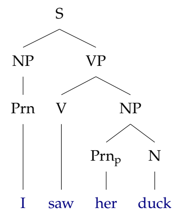
Derivation of sentence 'she saw a duck'

S ⇒ NP VP
NP ⇒ Prn
Prn ⇒ she
VP ⇒ V NP
V ⇒ saw
NP ⇒ Det N
Det ⇒ a
N ⇒ duck



Representations of a context-free parse tree

A parse tree:



A history of derivations:

- S ⇒NP VP
- NP ⇒Prn
- Prn ⇒I
- VP ⇒V NP
- V ⇒saw
- NP ⇒Prn_p N
- Prn_p ⇒her
- N ⇒duck

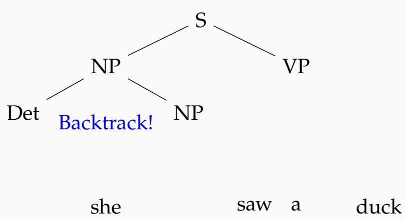
A sequence with (labeled) brackets

$$\left[_{\text{S}} \left[_{\text{NP}} [_{\text{Prn}}\ \text{I}] \right] \left[_{\text{VP}} [_{\text{V}}\ \text{saw}]\ [_{\text{NP}} [_{\text{Prn}_\text{p}}\ \text{her}]\ [_{\text{N}}\ \text{duck}]] \right] \right]$$

Parsing as search

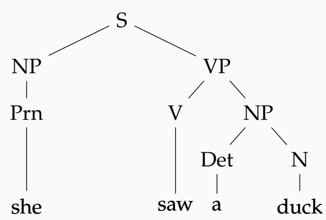
- Parsing can be seen as search constrained by the grammar and the input
- Top down: start from S, find the derivations that lead to the sentence
- Bottom up: start from the sentence, find series of derivations (in reverse) that leads to S
- Search can be depth first or breadth first for both cases

Parsing as search: top down



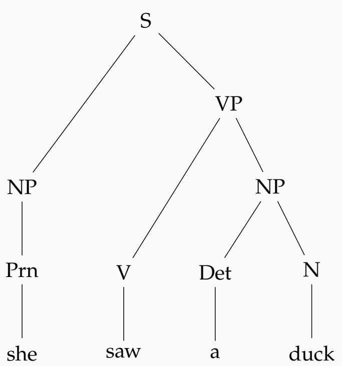
S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the

Parsing as search: top down



S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the

Parsing as search: bottom up



S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the

Problems with search procedures

- Top-down search considers productions incompatible with the input, and cannot handle left recursion
- Bottom-up search considers non-terminals that would never lead to S
- Repeated work because of backtracking

→ The result is exponential time complexity in the length of the sentence

Some of these problems can be solved using *dynamic programming*.

CKY algorithm

- The CKY (Cocke–Younger–Kasami), or CYK, parsing algorithm is a dynamic programming algorithm (Kasami 1965; Younger 1967; Cocke and Schwartz 1970)
- It processes the input *bottom up*, and saves the intermediate results on a *chart*
- Time complexity for *recognition* is $O(n^3)$ (with a space complexity of $O(n^2)$)
- It requires the CFG to be in *Chomsky normal form* (CNF)

Chomsky normal form (CNF)

- A CFG is in CNF, if the rewrite rules are in one of the following forms
  - $A \rightarrow B\ C$
  - $A \rightarrow a$

  where A, B, C are non-terminals and a is a terminal
- Any CFG can be converted to CNF
- Resulting grammar is *weakly equivalent* to the original grammar:
  - it generates/accepts the same language
  - but the derivations are different

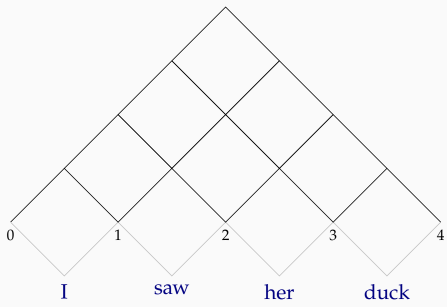
Converting to CNF: example

- For rules with > 2 RHS symbols
  S → Aux NP VP ⇒ S → Aux X
  X → NP VP
- For rules with < 2 RHS symbols
  NP → Prn ⇒ NP → she | her

S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the

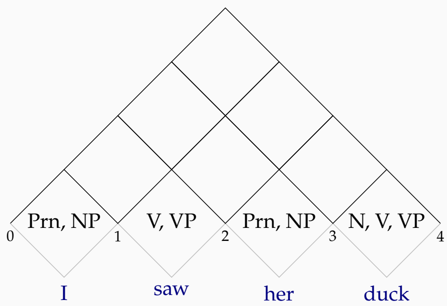
CKY demonstration

an ambiguous example



CKY demonstration

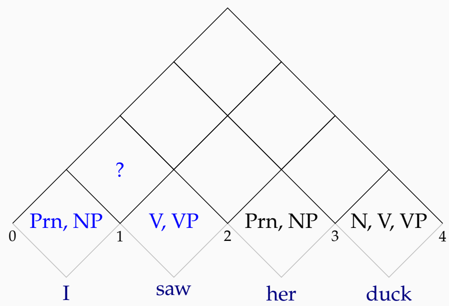
an ambiguous example



CKY demonstration

an ambiguous example

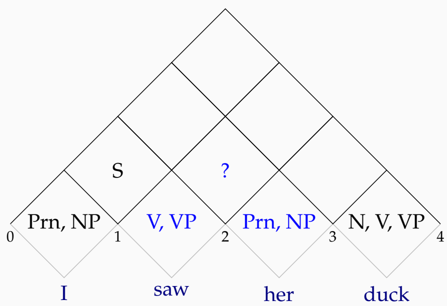
S → NP VP



CKY demonstration

an ambiguous example

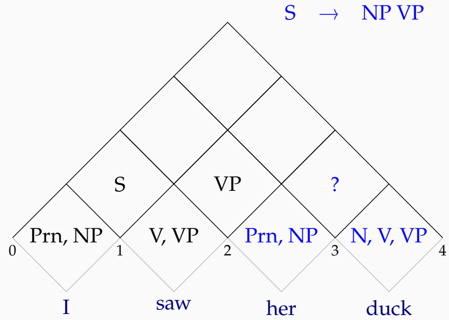
VP → V NP



CKY demonstration

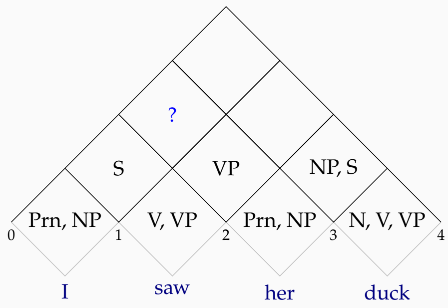
an ambiguous example

NP → Prn N
S → NP VP



CKY demonstration

an ambiguous example

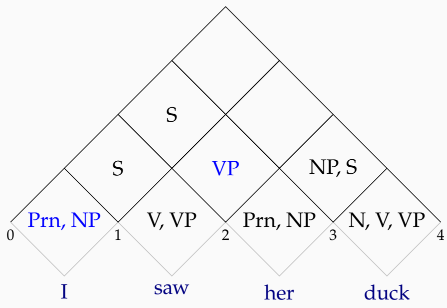


Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration

an ambiguous example

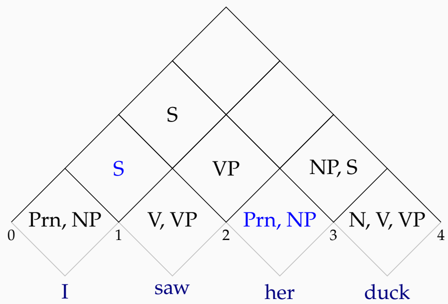
S → NP VP



Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration

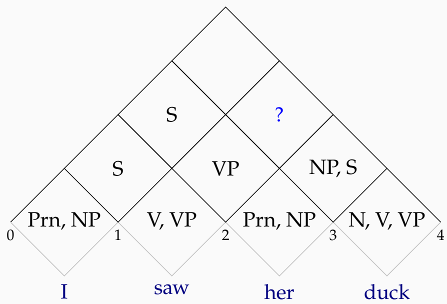
an ambiguous example



Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration

an ambiguous example

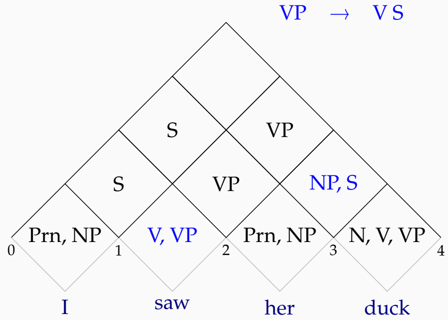


Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration

an ambiguous example

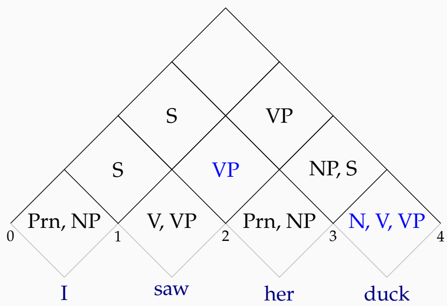
VP → V NP
VP → V S



Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration

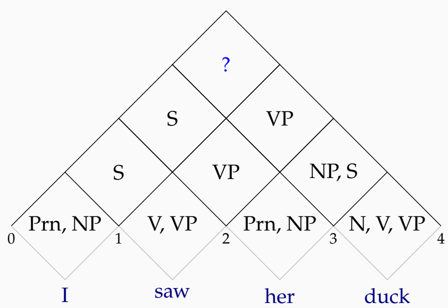
an ambiguous example



Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration

an ambiguous example

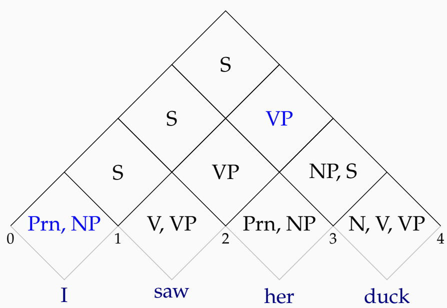


Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration

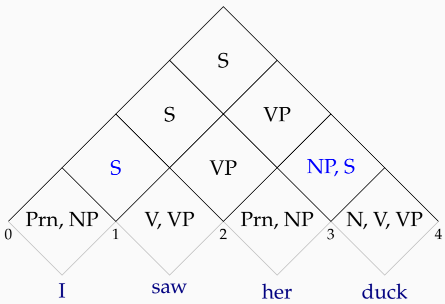
an ambiguous example

S → NP VP



CKY demonstration

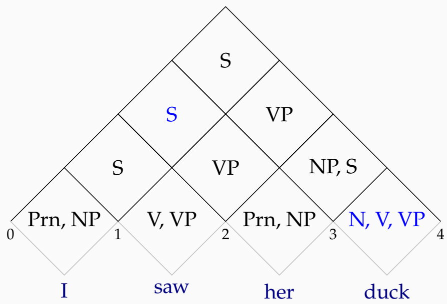
an ambiguous example



Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration

an ambiguous example



Ç. Çöltekin, SfS / University of Tübingen Summer Semester 2018 26 / 102

Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY demonstration: the chart

NP, Prn	S	S	S
	V, VP	VP	VP
		Prn	NP, S
			V, N, NP

$_0$ she $_1$ saw $_2$ her $_3$ duck $_4$

Chart is a 2-dimensional array, hence $O(n^2)$ space complexity.

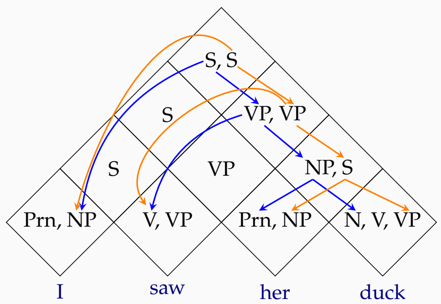
Grammars Constituency parsing Dependency grammars Dependency parsing Summary

Parsing vs. recognition

- We went through a recognition example
- Recognition accepts or rejects as sentence based on a grammar
- For parsing, we want to know the derivations that yielded a correct parse
- To recover parse trees, we
  - we follow the same procedure as recognition
  - add back links to keep track of the derivations

Grammars Constituency parsing Dependency grammars Dependency parsing Summary

Chart parsing example (CKY parsing)



Grammars Constituency parsing Dependency grammars Dependency parsing Summary

CKY summary

\+ CKY avoids re-computing the analyses by storing the earlier analyses (of sub-spans) in a table

– It still computes lower level constituents that are not allowd by the grammar

– CKY requires the grammar to be in CNF

- CKY has $O(n^3)$ recognition complexity
- For parsing we need to keep track of backlinks
- CKY can effciently store all possible parses in a chart
- Enumerating all possible parses have exponential complexity (worst case)

Grammars Constituency parsing Dependency grammars Dependency parsing Summary

Earley algorithm

- Earley algorithm is a top down parsing algorithm (Earley 1970)
- It allows arbitrary CFGs
- Keeps record of constituents that are
  - predicted using the grammar (top-down)
  - in-progress with partial evidence
  - completed based on input seen so far

  at every position in the input string
- Time complexity is $O(n^3)$

Grammars Constituency parsing Dependency grammars Dependency parsing Summary

Earley chart entries (states or items)

Earley chart entries are CF rules with a 'dot' on the RHS representing the state of the rule

- $A \rightarrow \bullet\alpha[i,i]$ predicted without any evidence (yet)
- $A \rightarrow \alpha \bullet \beta[i,j]$ partially matched
- $A \rightarrow \alpha\beta \bullet [i,j]$ completed, the non-terminal A is found in the given span

Earley algorithm: an informal sketch

1. Start at position 0, predict S
2. Predict all possible states (rules that apply)
3. Read a word
4. Update the table, advance the dot if possible
5. Go to step 2
6. If we have a completed S production at the end of the input, the input it recognized

Earley algorithm: three operations

Predictor adds all rules that are possible at the given state

Completer adds states from the earlier chart entries that match the completed state to the chart entry being processed, and advances their dot

Scanner adds a completed state to the next chart entry if the current category is a POS tag, and the word matches

Earley parsing example (chart[0])

0 she 1 saw 2 a 3 duck 4

state	rule	position	operation
0	γ →•S	[0,0]	initialization
1	S →•NP VP	[0,0]	predictor
2	S →• Aux NP VP	[0,0]	predictor
3	NP →•Det N	[0,0]	predictor
4	NP →•NP PP	[0,0]	predictor
5	NP →•Prn	[0,0]	predictor

S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the
Aux → does | has

Earley parsing example (chart[1])

0 she 1 saw 2 a 3 duck 4

state	rule	position	operation
6	Prn →she •	[0,1]	scanner
7	NP →Prn •	[0,1]	completer
8	S →NP •VP	[0,1]	completer
9	NP →NP •PP	[0,1]	completer
10	VP →•V NP	[1,1]	predictor
11	VP →•VP PP	[1,1]	predictor
12	PP →•Prp NP	[1,1]	predictor

S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the
Aux → does | has

Earley parsing example (chart[2])

0 she 1 saw 2 a 3 duck 4

state	rule	position	operation
13	V →saw •	[1,2]	scanner
14	VP →V •NP	[1,2]	completer
15	VP →V •	[1,2]	completer
16	NP →•Det N	[2,2]	predictor
17	NP →•NP PP	[2,2]	predictor
18	NP →•Prn	[2,2]	predictor
19	S →NP VP •	[0,2]	predictor

S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the
Aux → does | has

Earley parsing example (chart[3])

0 she 1 saw 2 a 3 duck 4

state	rule	position	operation
20	Det →a •	[2,3]	scanner
21	NP →Det •N	[2,3]	completer

S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the
Aux → does | has

Earley parsing example (chart[4])

0 she 1 saw 2 a 3 duck 4

state	rule	position	operation
22	N →duck •	[3,4]	scanner
23	NP →Det N •	[2,4]	completer
24	VP →V NP •	[1,4]	completer
25	S →NP VP •	[0,4]	completer

S → NP VP
S → Aux NP VP
NP → Det N
NP → Prn
NP → NP PP
VP → V NP
VP → V
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the
Aux → does | has

Summary: context-free parsing algorithms

- Naive search for parsing is intractable
- Dynamic programming algorithms allow polynomial time recognition
- Parsing may still be exponential in the worse case
- Ambiguity: CKY or Earley parse tables can represent ambiguity, but cannot say anything about which parse is the best

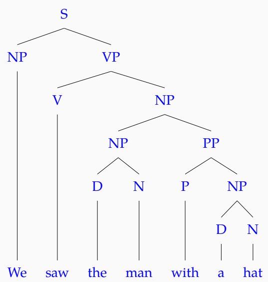
Pretty little girl's school (again)

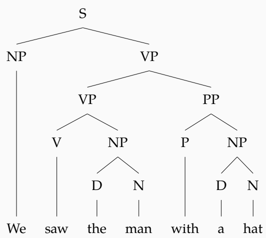
Cartoon Theories of Linguistics, SpecGram Vol CLIII, No 4, 2008. `http://specgram.com/CLIII.4/school.gif`

Some more examples

- Lexical ambiguity
  - She is looking for a match
  - We saw her duck
- Attachment ambiguity
  - I saw the man with a telescope
  - Panda eats bamboo shoots and leaves
- Local ambiguity (garden path sentences)
  - The horse raced past the barn fell
  - The old man the boats
  - Fat people eat accumulates

The task: choosing the most plausible parse





Statistical parsing

- Find the most plausible parse of an input string given all possible parses
- We need a scoring function, for each parse, given the input
- We typically use probabilities for scoring, task becomes finding the parse (or tree), t, given the input string $\boldsymbol{w}$

$$t_{best} = \arg\max_{t} P(t \mid \boldsymbol{w})$$

- Note that some ambiguities need a larger context than the sentence to be resolved correctly

Probabilistic context free grammars (PCFG)

A probabilistic context free grammar is specified by,

- Σ is a set of terminal symbols
- N is a set of non-terminal symbols
- S $\in N$ is a distinguished *start* symbol
- R is a set of rules of the form

$$A \rightarrow \alpha \quad [p]$$

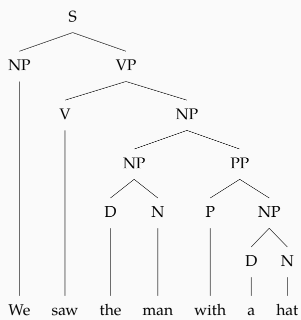
where A is a non-terminal, α is string of terminals and non-terminals, and p *is the probability associated with the rule*

- The grammar accepts a sentence if it can be derived from S with rules $R_1 \ldots R_k$
- *The probability of a parse* t *of input string* $\boldsymbol{w}$, $P(t \mid \boldsymbol{w})$, *corresponding to the derivation* $R_1 \ldots R_k$ *is*

$$P(t \mid \boldsymbol{w}) = \prod_1^k p_i$$

where p_i *is the probability of the rule* R_i

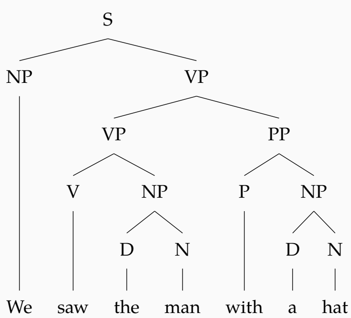
PCFG example (1)



S	→ NP VP	1.0
NP	→ D N	0.7
NP	→ NP PP	0.2
NP	→ We	0.1
VP	→ V NP	0.9
VP	→ VP PP	0.1
PP	→ P NP	1.0
N	→ hat	0.2
N	→ man	0.8
V	→ saw	1.0
P	→ with	1.0
D	→ a	0.6
D	→ the	0.4

$$P(t) = 1.0 \times 0.1 \times 0.9 \times 1.0 \times 0.2 \times 0.7 \times 0.4 \times 0.8 \times 1.0 \times 1.0 \times 0.7 \times 0.6 \times 0.2$$
$$= 0.000263424$$

PCFG example (2)



S	→ NP VP	1.0
NP	→ D N	0.7
NP	→ NP PP	0.2
NP	→ We	0.1
VP	→ V NP	0.9
VP	→ VP PP	0.1
PP	→ P NP	1.0
N	→ hat	0.2
N	→ man	0.8
V	→ saw	1.0
P	→ with	1.0
D	→ a	0.6
D	→ the	0.4

$$P(t) = 1.0 \times 0.1 \times 0.1 \times 0.9 \times 1.0 \times 0.7 \times 0.4 \times 0.8 \times 1.0 \times 1.0 \times 0.7 \times 0.6 \times 0.2$$
$$= 0.0001693440$$

Where do the rule probabilities come from?

- Supervised: estimate from a treebank, e.g., using maximum likelihood estimation
- Unsupervised: expectation-maximization (EM)

PCFGs - an interim summary

- PCFGs assign probabilities to parses based on CFG rules used during the parse
- PCFGs assume that the rules are independent
- PCFGs are generative models, they assign probabilities to $P(t, \boldsymbol{w})$, we can calcuate the probability of a sentence by

$$P(\boldsymbol{w}) = \sum_t P(t, \boldsymbol{w}) = \sum_t P(t)$$

What makes the difference in PCFG probabilities?

S	$\Rightarrow$ NP VP	1.0	S	$\Rightarrow$ NP VP	1.0
NP	$\Rightarrow$ We	0.1	NP	$\Rightarrow$ We	0.1
VP	$\Rightarrow$ VP PP	0.1	VP	$\Rightarrow$ V NP	0.7
VP	$\Rightarrow$ V NP	0.8	V	$\Rightarrow$ saw	1.0
V	$\Rightarrow$ saw	1.0	NP	$\Rightarrow$ NP PP	0.2
NP	$\Rightarrow$ D N	0.7	NP	$\Rightarrow$ D N	0.7
D	$\Rightarrow$ the	0.4	D	$\Rightarrow$ the	0.4
N	$\Rightarrow$ man	0.8	N	$\Rightarrow$ man	0.8
PP	$\Rightarrow$ P NP	1.0	PP	$\Rightarrow$ P NP	1.0
P	$\Rightarrow$ with	1.0	P	$\Rightarrow$ with	1.0
NP	$\Rightarrow$ D N	0.7	NP	$\Rightarrow$ D N	0.7
D	$\Rightarrow$ a	0.6	D	$\Rightarrow$ a	0.6
N	$\Rightarrow$ hat	0.2	N	$\Rightarrow$ hat	0.2

The parser's choice would not be affected by lexical items!

What is wrong with PCFGs?

- In general: the assumption of independence
- The parents affect the correct choice for children, for example, in English NP →Prn is more likely in the subject position
- The lexical units affect the correct decision, for example:
  - We eat the pizza with hands
  - We eat the pizza with mushrooms
- Additionally: PCFGs use local context, difficult to incorporate arbitrary/global features for disambiguation

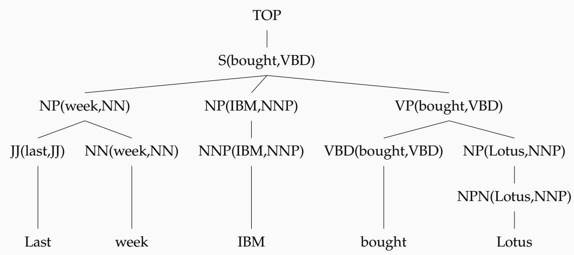
Solutions to PCFG problems

- Independence assumptions can be relaxed by either
  - Parent annotation
  - Lexicalization - Collins (1999)
- To condition on arbitrary/global information: disciriminative models - Charniak and Johnson (2005)
- Most practical PCFG parsers are lexicalized, and often use a re-ranker conditioning on other (global) features

Lexicalizing PCFGs

- Replace non-terminal X with X(h), where h is a tuple with the lexical word and its POS tag
- Now the grammar can capture (head-driven) lexical dependencies
- But number of nonterminals grow by $|V| \times |T|$
- Estimation becomes difficult (many rules, data sparsity)
- Some treebanks (e.g., Penn Treebank) do not annotate heads, they are automatically annotated (based on heuristics)

Example lexicalized derivation



Example rules:

TOP	→	S(bought,VBD)
S(bought,VBD)	→	NP(week,NN) NP(IBM,NNP) VP(bought,VBD)
VP(bought,VBD)	→	VBD(bought,VBD) NP(Lotus,NNP)
JJ(last,JJ)	→	Last

Evaluating the parser output

- A parser can be evaluated
  - extrinsically: based on it's effect on a task (e.g., machine translation) where it is used
  - intrinsically: based on the match with ideal parsing
- The typically evaluation (intrinsic) is based on a *gold standard* (GS)
- Exact match is often
  - very difficult to achieve (think about a 50-word newspaper sentence)
  - not strictly necessary (recovering parts of the parse can be useful for many purposes)

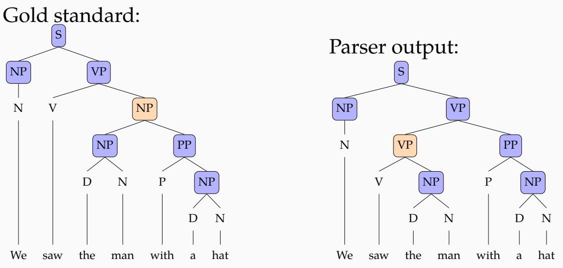
Parser evaluation metrics

- Common evaluation metrics are (PARSEVAL):
  - precision: the ratio of correctly predicted nodes
  - recall: the nodes (in GS) that are predicted correctly
  - f-measure: harmonic mean of precision and recall $\left(\frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}\right)$
- The measures can be
  - unlabled: the spans of the nodes are expected to match
  - labeled: the node label should also match
- Crossing brackets (or average non-crossing brackets)

(	We	(	saw	(	them	(	with	binoculars	))))
(	We	((	saw		them	)	(with	binoculars	)))

- Measures can be averaged per constituent (micro average), or over sentences (macro average)

PARSEVAL example



$$\text{precision} = \frac{6}{7} \quad \text{recall} = \frac{6}{7} \quad \text{f-measure} = \frac{6}{7}$$

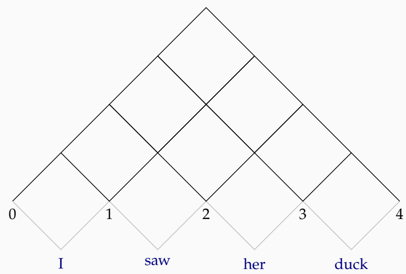
Problems with PARSEVAL metrics

- PARSEVAL metrics favor certain type of structures
  - Results are surprisingly well for flat tree structures (e.g., Penn treebank)
  - Results of some mistakes are catastrophic (e.g., low attachment)
- Not all mistakes are equally important for semantic distinctions
- Some alternatives:
  - Extrinsic evaluation
  - Evaluation based on extracted dependencies

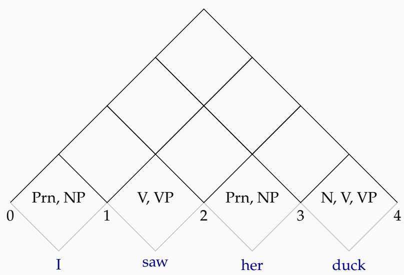
PCFG chart parsing

- Both CKY and Earley algorithms can be adapted to PCFG parsing
- CKY matches PCFG parsing quite well
  - to get the best parse, store the constituent with the highest probability in every cell of the chart
  - to get n-best best parse (beam search), store the n-best constituents in every cell in the chart

CKY for PCFG parsing



CKY for PCFG parsing



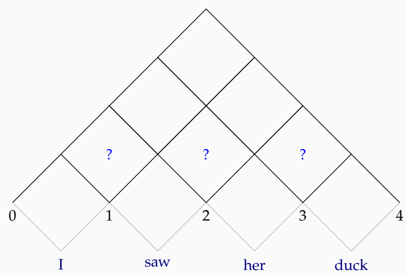
$P(\text{Prn}_{01}) = P(\text{Prn} \to \text{I})$ $\quad$ $P(\text{NP}_{01}) = P(\text{NP} \to \text{I})$

$P(\text{V}_{12}) = P(\text{V} \to \text{saw})$ $\quad$ $P(\text{VP}_{12}) = P(\text{VP} \to \text{saw})$

...

CKY for PCFG parsing

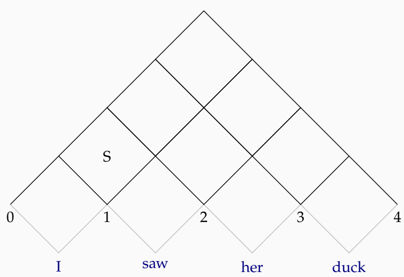
S → NP VP



$$P(\text{S}_{02} \Rightarrow \text{NP}_{01}\text{VP}_{12}) = P(\text{NP}_{01})P(\text{VP}_{12})P(\text{S} \to \text{NP VP})$$

CKY for PCFG parsing

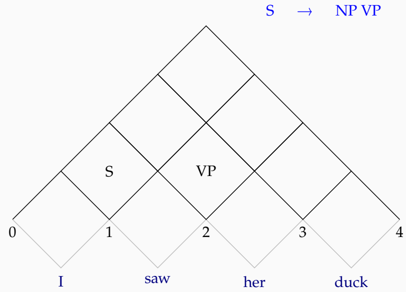
VP → V NP



$$P(\text{VP}_{13} \Rightarrow \text{V}_{12}\text{NP}_{23}) = P(\text{V}_{12})P(\text{NP}_{23})P(\text{VP} \to \text{V NP})$$

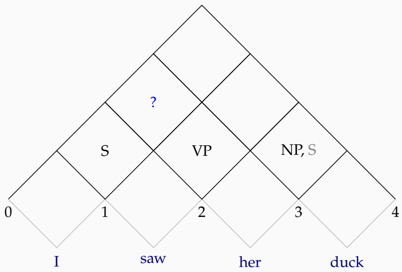
CKY for PCFG parsing

NP → Prn N
S → NP VP



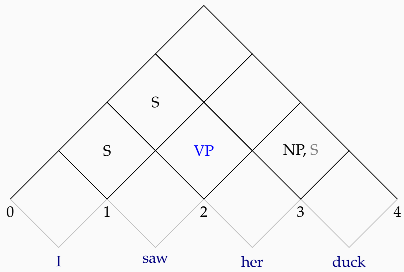
$$P(\text{NP}_{24} \Rightarrow \text{Prn}_{23}\text{N}_{34}) = P(\text{Prn}_{23})P(\text{N}_{34})P(\text{Prn} \to \text{Prn N})$$
$$>$$
$$P(\text{S}_{24} \Rightarrow \text{NP}_{23}\text{VP}_{34}) = P(\text{NP}_{23})P(\text{VP}_{34})P(\text{S} \to \text{NP VP})$$

CKY for PCFG parsing



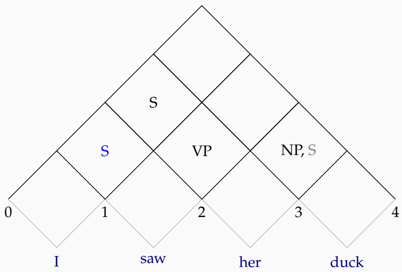
CKY for PCFG parsing

S → NP VP

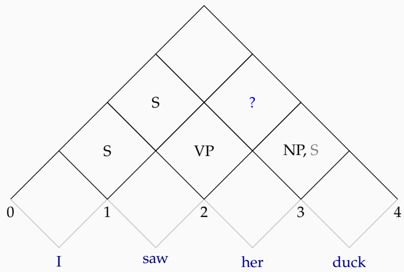


$$P(S_{03} \Rightarrow NP_{01} VP_{23}) = P(NP_{01}) P(VP_{13}) P(S \rightarrow NP\ VP)$$

CKY for PCFG parsing

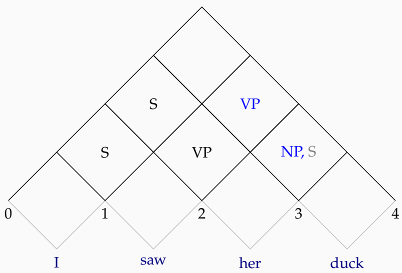


CKY for PCFG parsing



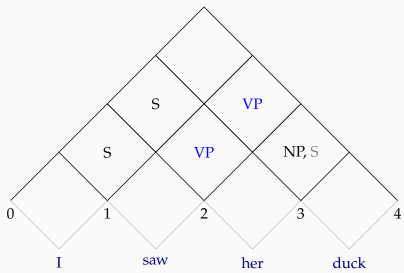
CKY for PCFG parsing

VP → V NP
VP → V S

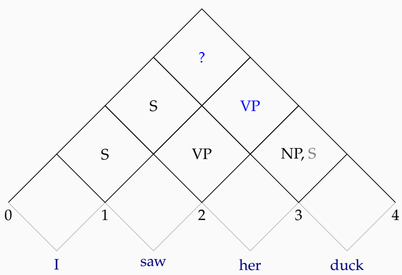


$$P(VP_{14} \Rightarrow V_{12} NP_{24}) = P(V_{12}) P(NP_{24}) P(VP \rightarrow V\ NP)$$

CKY for PCFG parsing

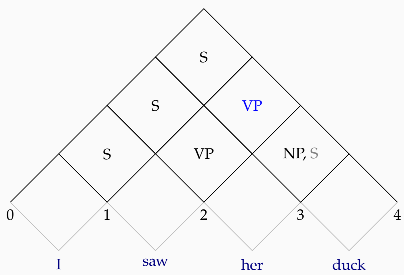


CKY for PCFG parsing



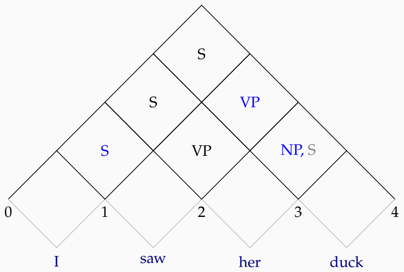
CKY for PCFG parsing

S → NP VP

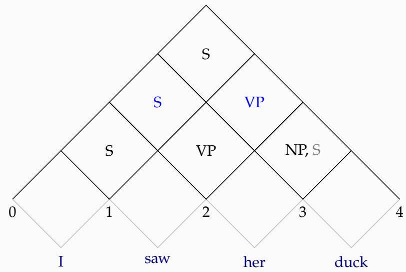


$$P(S_{14} \Rightarrow NP_{01} VP_{14}) = P(NP_{01}) P(VP_{14}) P(S \rightarrow NP\ VP)$$

CKY for PCFG parsing

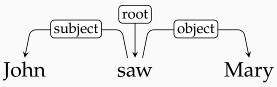


CKY for PCFG parsing



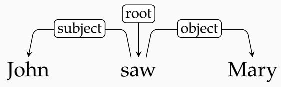
Dependency grammars

- Dependency grammars gained popularity in (particularly in computational) linguistics rather recently, but their roots can be traced back to a few thousand years (modern dependency grammars are attributed to Tesnière 1959)
- The main idea is capturing the relation between the words, rather than grouping them into (abstract) constituents



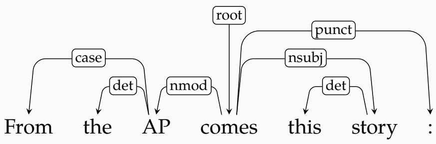
Note: like constituency grammars, we will not focus on a particular dependency formalism, but discuss it in general in relation to parsing.

Properties of dependency grammars



- The structure of the sentence is represented by asymmetric binary links between *lexical items*
- Each relation defines one of the words as the *head* and the other as *dependent*
- The links (relations) have labels (dependency types)
- Most dependency grammar require each word to have only a single head

A more realistic example



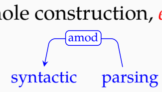
How to determine heads

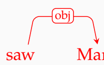
1. *Head* (H) determines the syntactic category of the *construction* (C) and can often replace C
2. H determines the semantic category of C; the *dependent* (D) gives semantic specification
3. H is obligatory, D may be optional
4. H selects D and determines whether D is obligatory or optional
5. The form and/or position of dependent is determined by the head
6. The form of D depends on H
7. The linear position of D is specified with reference to H

(from Kübler, McDonald, and Nivre 2009, p.3–4)

Issues with head assignment and dependency labels

- Like the tests for constituency, determining heads are not always straightforward
- A construction is called *endocentric* if the head can replace the whole construction, *exocentric* otherwise

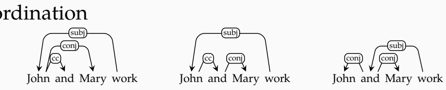




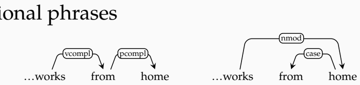
- It is often unclear whether dependency labels encode syntactic or semantic functions

Some tricky constructions

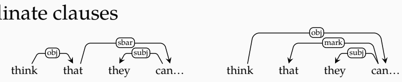
- Coordination



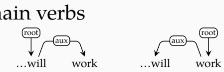
- Prepositional phrases



- Subordinate clauses



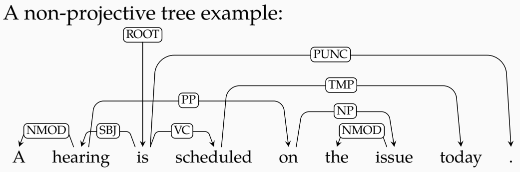
- Auxiliaries vs. main verbs



Projective vs. non-projective dependencies

- If a dependency graph has no crossing edges, it is said to be *projective*, otherwise *non-projective*
- Non-projectivity stems from long-distance dependencies and free word order

A non-projective tree example:



(tree reproduced from McDonald and Satta 2007)

Parsing with dependency grammars

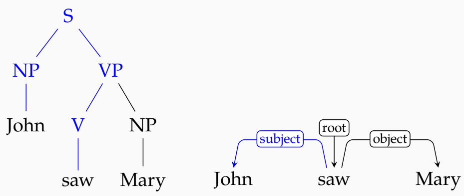
- Projective parsing can be done in polynomial time
- Non-projective parsing is NP-hard (without restrictions)
- For both, it is a common practice to use greedy (e.g., linear time) algorithms

Dependency vs. constituency

- Constituency grammars are based on units formed by a group of lexical items (constituents or phrases)
- Dependency grammars model binary head–dependent relations between words
- Most of the theory of parsing is developed with constituency grammars
- Dependency grammars has recently become more popular in CL
- Note that many formalisms and treebanks follow a hybrid approach, using ideas from both

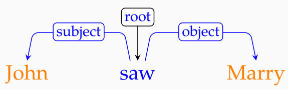
Conversion between constituencies and dependencies

- Although non-trivial, conversion between dependency and consitituency annotation is possible
- On can take the path between two words as a dependency relation



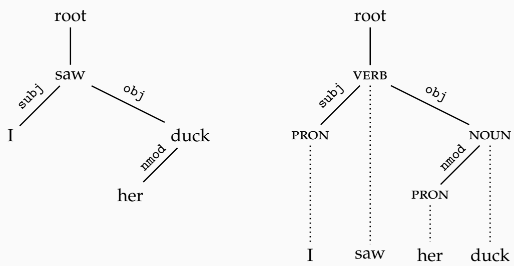
- The conversion from constituencies to dependencies is a common practice in the field

Dependency grammars



- No constituents, units of syntactic structure are words
- The structure of the sentence is represented by asymmetric binary relations between syntactic units
- The links (relations) have labels (dependency types)
- Each relation defines one of the words as the head and the other as dependent
- Often an artificial *root* node is used for computational convenience

Dependency grammars: notational variation



Dependency grammar: definition

A dependency grammar is a tuple (V, A)

V is a set of nodes corresponding to the (syntactic) words (we implicitly assume that words have indexes)

A is a set of arcs of the form (w_i, r, w_j) where

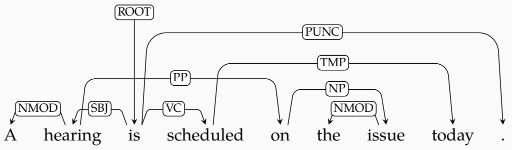
- w_i $\in V$ is the head
- r is the type of the relation (arc label)
- w_j $\in V$ is the dependent

This defines a directed graph.

Dependency grammars: common assumptions

- Every word has a single head
- The dependency graphs are acyclic
- The graph is connected
- With these assumptions, the representation is a tree
- Note that these assumptions are not universal but common for dependency parsing

Dependency grammars: projectivity

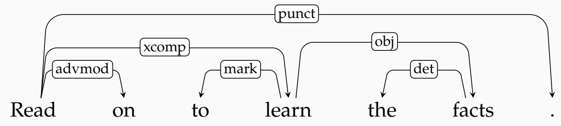


- If a dependency graph has no crossing edges, it is said to be *projective*, otherwise *non-projective*
- Non-projectivity stems from long-distance dependencies and free word order
- Projective dependency trees can be represented with context-free grammars
- In general, projective dependencies are parseable more efficiently

CONLL-X/U format for dependency annotation

Single-head assumption allows flat representation of dependency trees

```
1   Read  read  VERB  VB  Mood=Imp|VerbForm=Fin 0 root
2   on    on    ADV   RB  _                     1 advmod
3   to    to    PART  TO  _                     4 mark
4   learn learn VERB  VB  VerbForm=Inf          1 xcomp
5   the   the   DET   DT  Definite=Def          6 det
6   facts fact  NOUN  NNS Number=Plur           4 obj
7   .     .     PUNCT .   _                     1 punct
```



example from English Universal Dependencies treebank

Dependency parsing

- Dependency parsing has many similarities with context-free parsing (e.g., trees)
- They also have some different properties (e.g., number of edges and depth of trees are limited)
- Dependency parsing can be
  - grammar-driven (hand crafted rules or constraints)
  - data-driven (rules/model is learned from a treebank)
- There are two main approaches:

  Graph-based similar to context-free parsing, search for the best tree structure

  Transition-based similar to shift-reduce parsing (used for programming language parsing), but using greedy search for the best transition sequence

Grammar-driven dependency parsing

- Grammar-driven dependency parsers typically based on
  - lexicalized CF parsing
  - constraint satisfaction problem
    - start from fully connected graph, eliminate trees that do not satisfy the constraints
    - exact solution is intractable, often employ heuristics, approximate methods
    - sometimes 'soft', or weighted, constraints are used
  - Practical implementations exist
- Our focus will be on data-driven methods

Transition based parsing

- Inspired by shift-reduce parsing, single pass over the input
- Use a stack and a buffer of unprocessed words
- Parsing as predicting a sequence of transitions like

  LEFT-ARC: mark current word as the head of the word on top of the stack

  RIGHT-ARC: mark current word as a dependent of the word on top of the stack

  SHIFT: push the current word to the stack
- Algorithm terminates when all words in the input are processed
- The transitions are not naturally deterministic, best transition is predicted using a machine learning method

(Yamada and Matsumoto 2003; Nivre, Hall, and Nilsson 2004)

A typical transition system

$$(\underbrace{\sigma | \overbrace{w_i}^{\text{stack top}}}_{\text{stack}},\ \underbrace{\overbrace{w_j}^{\text{next word}} | \beta}_{\text{buffer}},\ \underbrace{A}_{\text{arcs}})$$

LEFT-ARC$_r$: $(\sigma|w_i, w_j|\beta, A) \Rightarrow (\sigma \quad, w_j|\beta, A \cup \{(w_j, r, w_i)\})$
- pop w_i,
- add arc (w_j, r, w_i) to A (keep w_j in the buffer)

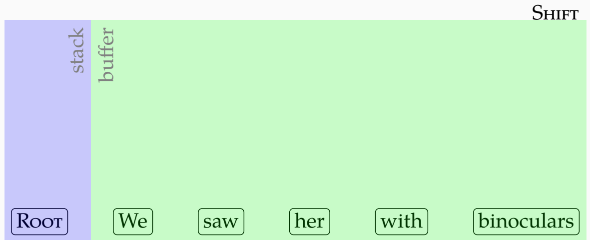
RIGHT-ARC$_r$: $(\sigma|w_i, w_j|\beta, A) \Rightarrow (\sigma \quad, w_i|\beta, A \cup \{(w_i, r, w_j)\})$
- pop w_i,
- add arc (w_i, r, w_j) to A,
- move w_i to the buffer

SHIFT: $(\sigma \quad, w_j|\beta, A) \Rightarrow (\sigma|w_j, \quad \beta, A)$
- push w_j to the stack
- remove it from the buffer

(Kübler, McDonald, and Nivre 2009, p.23)

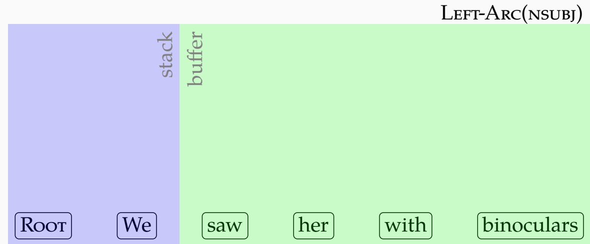
Transition based parsing: example

SHIFT

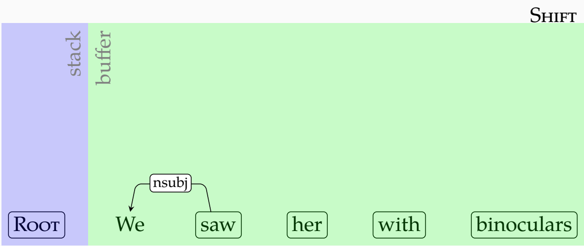


Transition based parsing: example

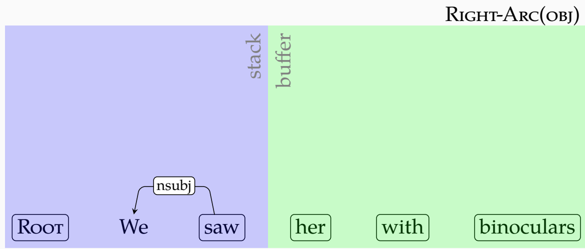
LEFT-ARC(NSUBJ)



Transition based parsing: example

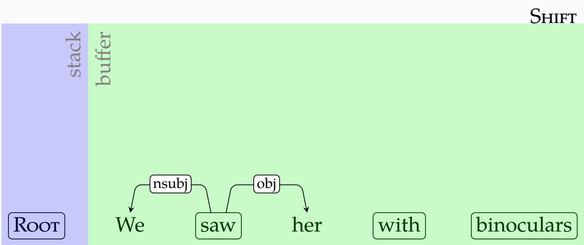


Transition based parsing: example

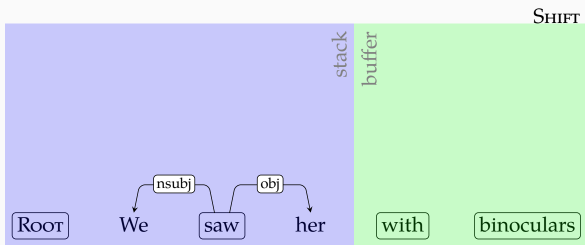


Note: We need Shift for NP attachment.

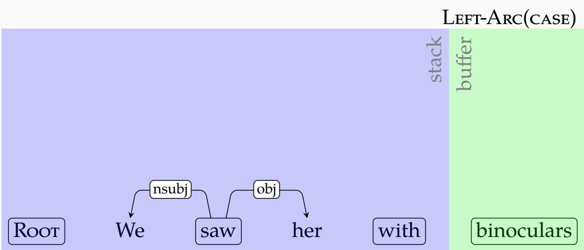
Transition based parsing: example



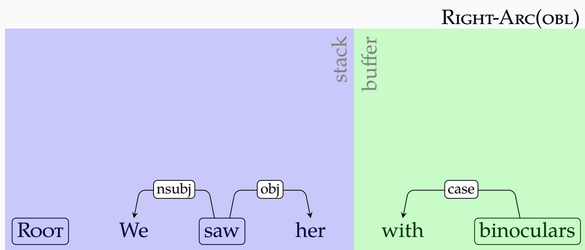
Transition based parsing: example



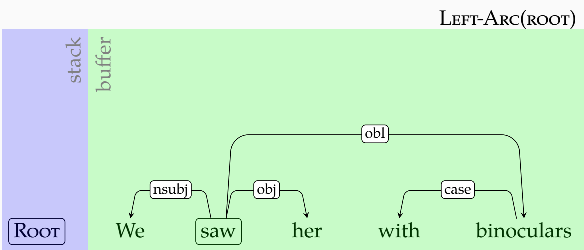
Transition based parsing: example



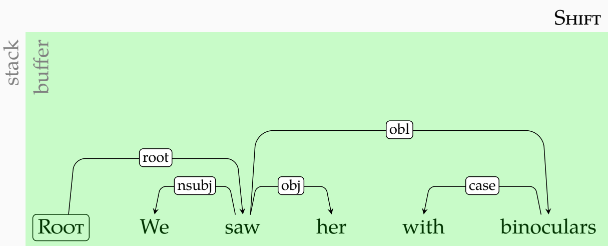
Transition based parsing: example



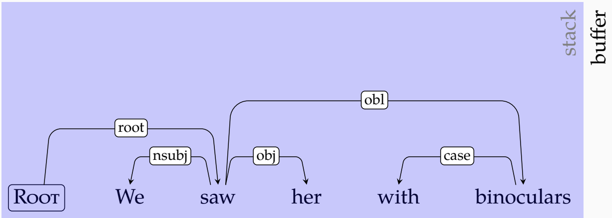
Transition based parsing: example



Transition based parsing: example



Transition based parsing: example



Making transition decisions

- In classical shift-reduce parsing the actions are deterministic
- In transition-based dependency parsing, we need to choose among all possible transitions
- The typical method is to train a (discriminative) classifier on features extracted from gold-standard *transition sequences*
- Almost any machine learning method method is applicable. Common choices include
  - Memory-based learning
  - Support vector machines
  - (Deep) neural networks

Features for transition-based parsing

- The features come from the parser configuration, for example
  - The word at the top of the stack, (peeking towards the bottom of the stack is also fine)
  - The first/second word on the buffer
  - Right/left dependents of the word on top of the stack/buffer
- For each possible 'address', we can make use of features like
  - Word form, lemma, POS tag, morphological features, word embeddings
  - Dependency relations – (w_i, r, w_j) triples
- Note that for some 'address'–'feature' combinations and in some configurations the values may be missing

The training data

- We want features like,
  - `lemma[Stack] = duck`
  - `POS[Stack] = NOUN`
  - ...
- But treebank gives us:

```
1   Read  read  VERB  VB  Mood=Imp|VerbForm=Fin 0 root
2   on    on    ADV   RB  _                     1 advmod
3   to    to    PART  TO  _                     4 mark
4   learn learn VERB  VB  VerbForm=Inf          1 xcomp
5   the   the   DET   DT  Definite=Def          6 det
6   facts fact  NOUN  NNS Number=Plur           4 obj
7   .     .     PUNCT .   _                     1 punct
```

- The treebank has the outcome of the parser, but none of our features.

The training data

- The features for transition-based parsing have to be from *parser configurations*
- The data (treebanks) need to be preprocessed for obtaining the training data
- Construct a transition sequence by parsing the sentences, and using treebank annotations (the set A) as an 'oracle'
- Decide for

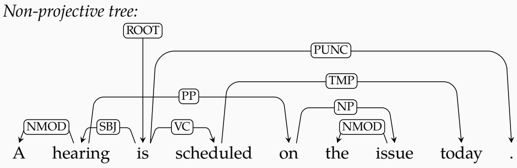
  LEFT-ARC$_r$ if $(\beta[0], r, \sigma[0]) \in A$
  RIGHT-ARC$_r$ if $(\sigma[0], r, \beta[0]) \in A$ and all dependents of $\beta[0]$ are attached
  SHIFT otherwise
- There may be multiple sequences that yield the same dependency tree, the above defines a 'canonical' transition sequence

Non-projective parsing

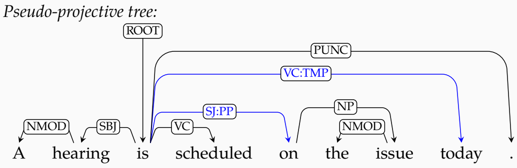
- The transition-based parsing we defined so far works only for projective dependencies
- One way to achieve (limited) non-projective parsing is to add special LEFT-ARC and RIGHT-ARC transitions to/from non-top words from the stack
- Another method is pseudo-projective parsing:
  - preprocessing to 'projectivize' the trees before training
    - The idea is to attach the dependents to a higher level head that preserves projectivity, while marking it on the new dependency label
  - postprocessing for restoring the projectivity after parsing
    - Re-introduce projectivity for the marked dependencies

Pseudo-projective parsing

Non-projective tree:



Pseudo-projective tree:



Transition based parsing: summary/notes

- Linear time, greedy parsing
- Can be extended to non-projective dependencies
- One can use arbitrary features,
- We need some extra work for generating gold-standard transition sequences from treebanks
- Early errors propagate, transition-based parsers make more mistakes on long-distance dependencies
- The greedy algorithm can be extended to beam search for better accuracy (still linear time complexity)

Graph-based parsing: preliminaries

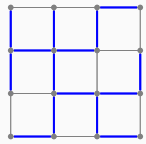
- Enumerate all possible dependency trees
- Pick the best scoring tree
- Features are based on limited parse history (like CFG parsing)
- Two well-known flavors:
  - Maximum (weight) spanning tree (MST)
  - Chart-parsing based methods

Eisner 1996; McDonald et al. 2005

MST parsing: preliminaries

Spanning tree of a graph

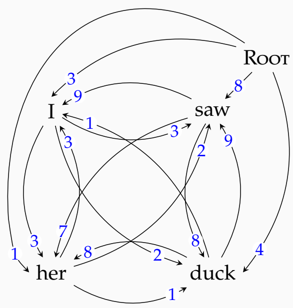
- Spanning tree of a connected graph is a sub-graph which is a tree and traverses all the nodes
- For fully-connected graphs, the number of spanning trees are exponential in the size of the graph
- The problem is well studied
- There are efficient algorithms for enumerating and finding the optimum spanning tree on weighted graphs



MST algorithm for dependency parsing

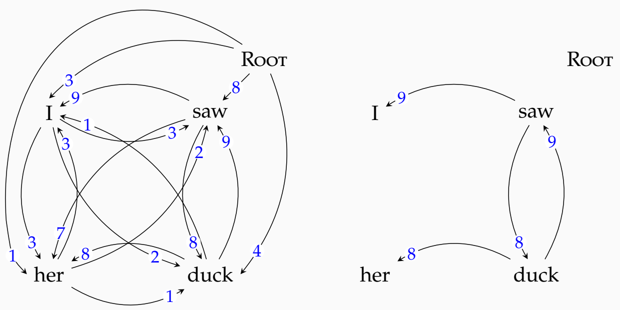
- For directed graphs, there is a polynomial time algorithm that finds the minimum/maximum spanning tree (MST) of a fully connected graph (Chu-Liu-Edmonds algorithm)
- The algorithm starts with a dense/fully connected graph
- Removes edges until the resulting graph is a tree

MST example



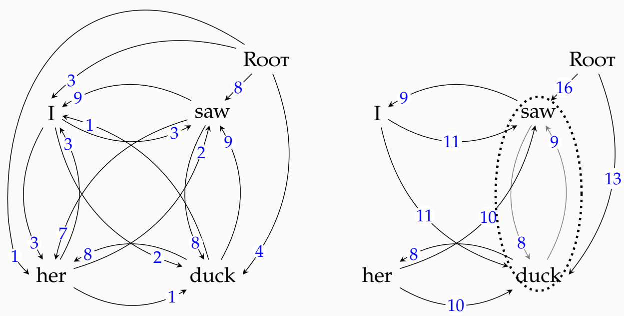
For each node select the incoming arc with highest weight

MST example



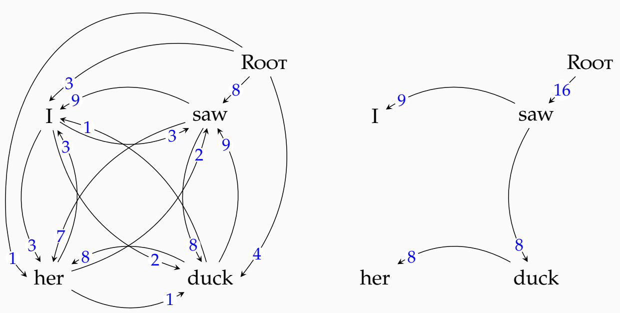
Detect the cycles, contract them to a 'single node'

MST example



Pick the best arc into the combined node, break the cycle

MST example



Once all cycles are eliminated, the result is the MST

Properties of the MST parser

- The MST parser is non-projective
- There is an alrgorithm with $O(n^2)$ time complexity (Tarjan 1977)
- The time complexity increases with typed dependencies (but still close to quadratic)
- The weights/parameters are associated with edges (often called 'arc-factored')
- We can learn the arc weights directly from a treebank
- However, it is difficult to incorporate non-local features

CKY for dependency parsing

- The CKY algorithm can be adapted to projective dependency parsing
- For a naive implementation the complexity increases drastically $O(n^6)$
  - Any of the words within the span can be the head
  - Inner loop has to consider all possible splits
- For projective parsing, the observation that the left and right dependents of a head are independently generated reduces the comlexity to $O(n^3)$

(Eisner 1997)

Non-local features

- The graph-based dependency parsers use edge-based features
- This limits the use of more global features
- Some extensions for using 'more' global features are possible
- This often leads non-projective parsing to become intractable

External features

- For both type of parsers, one can obtain features that are based on unsupervised methods such as
  - clustering
  - dense vector representations (embeddings)
  - alignment/transfer from bilingual corpora/treebanks

(Koo, Carreras, and Collins 2008)

Errors from different parsers

- Different parsers make different errors
  - Transition based parsers do well on local arcs, worse on long-distance arcs
  - Graph based parsers tend to do better on long-distance dependencies
- Parser combination is a good way to combine the powers of different models. Two common methods
  - Majority voting: train parsers separately, use the weighted combination of their results
  - Stacking: use the output of a parser as features for another

(McDonald and Satta 2007; Sagae and Lavie 2006; Nivre and McDonald 2008)

Evaluation metrics for dependency parsers

- Like CF parsing, exact match is often too strict
- *Attachment score* is the ratio of words whose heads are identified correctly.
  - *Labeled attachment score* (LAS) requires the dependency type to match
  - *Unlabeled attachment score* (UAS) disregards the dependency type
- *Precision/recall/F-measure* often used for quantifying success on identifying a particular dependency type

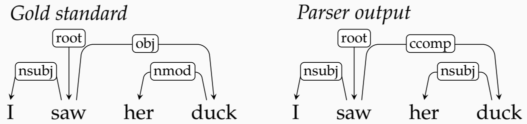
precision is the ratio of correctly identified dependencies (of a certain type)

recall is the ratio of dependencies in the gold standard that parser predicted correctly

f-measure is the harmonic mean of precision and recall $\left(\frac{2\times\text{precision}\times\text{recall}}{\text{precision}+\text{recall}}\right)$

Evaluation example

Gold standard *Parser output*



UAS	100%	
LAS	50%	
Precision$_{nsubj}$	50%	
Recall$_{nsubj}$	100%	
Precision$_{obj}$	0%	(assumed)
Recall$_{obj}$	0%	

Averaging evaluation scores

- As in context-free parsing, average scores can be

  macro-average or sentence-based

  micro-average or word-based
- Consider a two-sentence test set with

	words	correct
sentence 1	30	10
sentence 2	10	10

  - word-based average attachment score: 50% (20/40)
  - sentence-based average attachment score: 66% ((1 + 1/3)/2)

Dependency parsing: summary

- Dependency relations are often semantically easier to interpret
- It is also claimed that dependency parsers are more suitable for parsing free-word-order languages
- Dependency relations are between words, no phrases or other abstract nodes are postulated
- Two general methods:

  transition based greedy search, non-local features, fast, less accurate

  graph based exact search, local features, slower, accurate (within model limitations)
- Combination of different methods often result in better performance
- Non-projective parsing is more difficult
- Most of the recent parsing research has focused on better machine learning methods (mainly using neural networks)

Next

Mon/Fri Vector representations

Where to go from here?

- Textbook includes good coverage of constituency grammars and parsing, online 3rd edition includes a chapter on dependency parsing as well
- The book by Kübler, McDonald, and Nivre (2009) is an accessible introduction to (statistical) dependency parsing
- For more on linguistic and mathematical foundations of parsing:
  - Müller (2016) is a new open-source text book on Grammar formalisms.
  - Aho and Ullman (1972) is the classical reference (available online) for parsing (programming languages) and also includes discussion of grammar classes in the Chomsky hierarchy. A more up-to-date alternative is Aho, Lam, et al. (2007).

Where to go from here? (cont.)

- There is a brief introductory section on dependency grammars in Kübler, McDonald, and Nivre (2009), for a classical reference see Tesnière (2015), English translation of the original version (Tesnière 1959).

Pointers to some treebanks

Treebanks are the main resource for statistical parsing. A few treebank-related resources to have a look at until next time:

- Universal dependencies project, documentation, treebanks: `http://universaldependencies.org/`
- Tübingen treebanks:
  - TüBa-D/Z written German
  - TüBa-D/S spoken German
  - TüBa-E/S spoken English
  - TüBa-J/S spoken Japanese

  available from `http://www.sfs.uni-tuebingen.de/en/ascl/resources/corpora.html`
- TüNDRA - a treebank search and visualization application with the above treebanks and few more
  - Main version: `https://weblicht.sfs.uni-tuebingen.de/Tundra/`
  - New version (beta): `https://weblicht.sfs.uni-tuebingen.de/tundra-beta/`

CKY algorithm

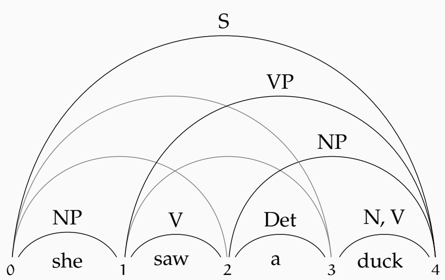
```
function CKY(words, grammar)
    for j ← 1 to LENGTH(words) do
        table[j − 1, j] ← {A|A → words[j] ∈ grammar}
        for i ← j − 1 downto 0 do
            for k ← i + 1 to j − 1 do
                table[i, j] ←   table[i, j] ∪
                              {A|A → BC ∈ grammar and
                                 B ∈ table[i, k] and
                                 C ∈ table[k, j]}
    return table
```


Even more examples

(newspaper headlines)

- FARMER BILL DIES IN HOUSE
- TEACHER STRIKES IDLE KIDS
- SQUAD HELPS DOG BITE VICTIM
- BAN ON NUDE DANCING ON GOVERNOR'S DESK
- PROSTITUTES APPEAL TO POPE
- KIDS MAKE NUTRITIOUS SNACKS
- DRUNK GETS NINE MONTHS IN VIOLIN CASE
- MINERS REFUSE TO WORK AFTER DEATH

Another CKY demonstration: spans



S → NP VP
S → Aux X
X → NP VP
NP → Det N
NP → she | her
NP → NP PP
VP → V NP
VP → duck | saw | ...
VP → VP PP
PP → Prp NP
N → duck
N → park
N → parks
V → duck
V → ducks
V → saw
Prn → she | her
Prp → in | with
Det → a | the