Natural Language Processing
Lecture 07 Syntax Parsing; PCFG Parsing; Dependency Parsing; Parsing with NNs

Qun Liu, Valentin Malykh
Huawei Noah's Ark Lab

Spring 2020
A course delivered at MIPT, Moscow

Content

1. Grammars and syntax parsing
2. Parsing with context free grammars
3. Parsing with probabilistic context free grammars
4. Dependency parsing
5. Parsing with neural networks

Content

1 Grammars and syntax parsing

2 Parsing with context free grammars

3 Parsing with probabilistic context free grammars

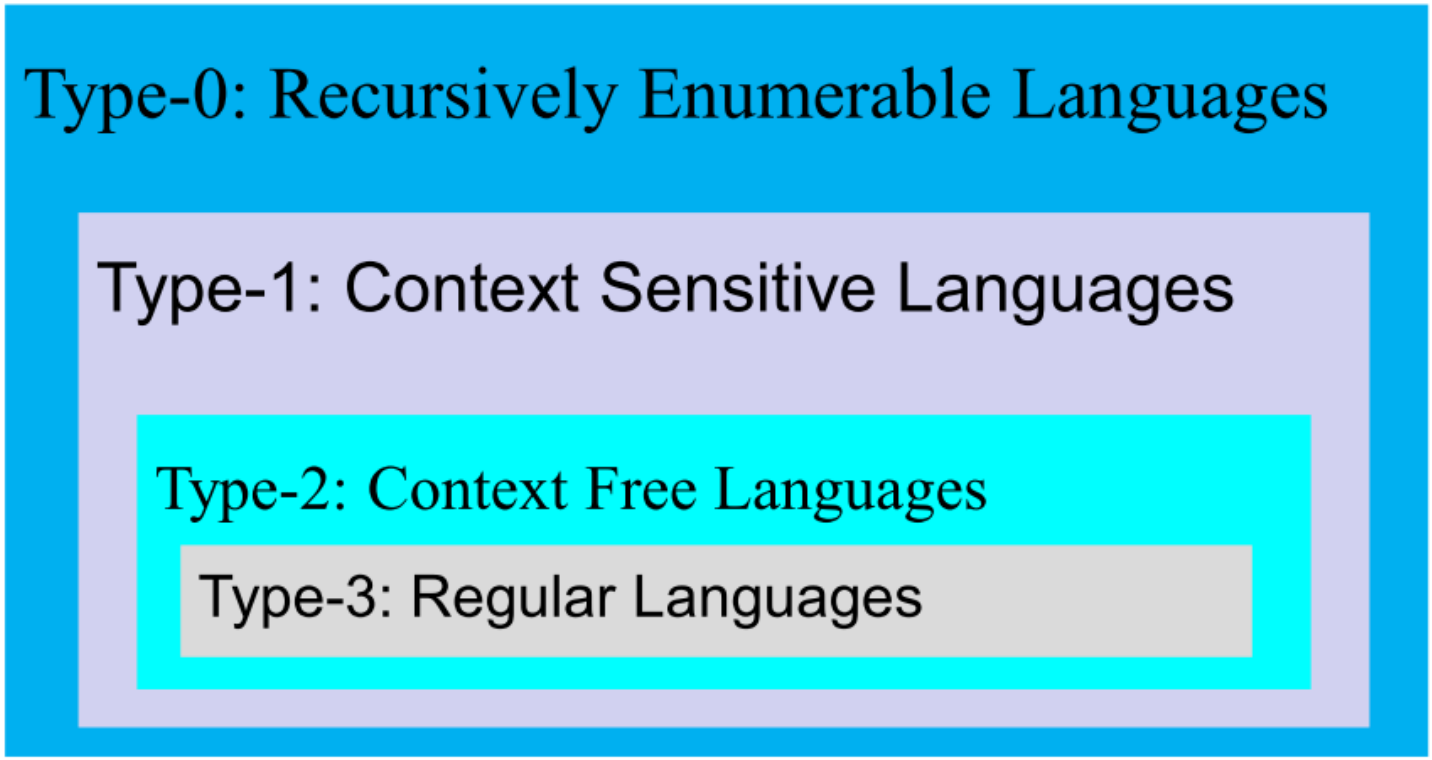
4 Dependency parsing

5 Parsing with neural networks

Chomsky hierarchy of grammars (Recap)

Grammar	Languages	Production Rules	Examples
Type 0	Recursively Enumerable	$\alpha A \beta \rightarrow \delta$	
Type 1	Context Sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$L = \{ a^n b^n c^n \mid n > 0 \}$
Type 2	Context Free	$A \rightarrow \alpha$	$L = \{ a^n b^n \mid n > 0 \}$
Type 3	Regular	$A \rightarrow a$ or $A \rightarrow aB$	$L = \{ a^n \mid n > 0 \}$

Chomsky hierarchy (Recap)



Set inclusions described by the Chomsky hierarchy

Desirable Properties of a Grammar

Chomsky specified two properties that make a grammar "interesting and satisfying":

- It should be a finite specification of the strings of the language, rather than a list of its sentences.
- It should be revealing, in allowing strings to be associated with meaning (semantics) in a systematic way.

We can add another desirable property:

- It should capture structural and distributional properties of the language. (E.g. where heads of phrases are located; how a sentence transforms into a question; which phrases can float around the sentence.)

Desirable Properties of a Grammar

- Context-free grammars (CFGs) provide a pretty good approximation.
- Some features of NLs are more easily captured using mildly context-sensitive grammars, as well see later in the course.
- There are also more modern grammar formalisms that better capture structural and distributional properties of human languages. (E.g. combinatory categorial grammar.)
- Programming language grammars (such as the ones used with compilers, like LL(1)) aren't enough for NLs.

A Tiny Fragment of English

Let's say we want to capture in a grammar the structural and distributional properties that give rise to sentences like:

A duck walked in the park.	NP,V,PP
The man walked with a duck.	NP,V,PP
You made a duck.	Pro,V,NP
You made her duck.	? Pro,V,NP
A man with a telescope saw you.	NP,PP,V,Pro
A man saw you with a telescope.	NP,V,Pro,PP
You saw a man with a telescope.	Pro,V,NP,PP

We want to write grammatical rules that generate these phrase structures, and lexical rules that generate the words appearing in them.

Grammar for the Tiny Fragment of English

Grammar G1 generates the sentences on the previous slide:

Grammatical rules	**Lexical rules**
S → NP VP	Det → a | the | her (determiners)
NP → Det N	N → man | park | duck | telescope (nouns)
NP → Det N PP	Pro → you (pronoun)
NP → Pro	V → saw | walked | made (verbs)
VP → V NP PP	Prep → in | with | for (prepositions)
VP → V NP	
VP → V	
PP → Prep NP	

Context-free grammars: formal definition

A context-free grammar (CFG) $\mathcal{G}$ consists of

- a finite set N of non-terminals,
- a finite set Σ of terminals, disjoint from N,
- a finite set P of productions of the form $X \rightarrow \alpha$, where $X \in N$, $\alpha \in (N \cup \Sigma)^*$,
- a choice of start symbol $S \in N$.

A sentential form is any sequence of terminals and nonterminals that can appear in a derivation starting from the start symbol.

Formal definition: The set of sentential forms derivable from $\mathcal{G}$ is the smallest set $\mathcal{S}(\mathcal{G}) \subseteq (N \cup \Sigma)^*$ such that

- $S \in \mathcal{S}(\mathcal{G})$
- if $\alpha X \beta \in \mathcal{S}(\mathcal{G})$ and $X \to \gamma \in P$, then $\alpha\gamma\beta \in \mathcal{S}(\mathcal{G})$.

The language associated with grammar is the set of sentential forms that contain only terminals.

Formal definition: The language associated with $\mathcal{G}$ is defined by $\mathcal{L}(\mathcal{G}) = \mathcal{S}(\mathcal{G}) \cap \Sigma^*$

A sentential form is any sequence of terminals and nonterminals that can appear in a derivation starting from the start symbol.

Formal definition: The set of sentential forms derivable from $\mathcal{G}$ is the smallest set $\mathcal{S}(\mathcal{G}) \subseteq (N \cup \Sigma)^*$ such that

- $S \in \mathcal{S}(\mathcal{G})$
- if $\alpha X \beta \in \mathcal{S}(\mathcal{G})$ and $X \rightarrow \gamma \in P$, then $\alpha\gamma\beta \in \mathcal{S}(\mathcal{G})$.

The language associated with grammar is the set of sentential forms that contain only terminals.

Formal definition: The language associated with $\mathcal{G}$ is defined by $\mathcal{L}(\mathcal{G}) = \mathcal{S}(\mathcal{G}) \cap \Sigma^*$

A language $L \subseteq \Sigma^*$ is defined to be context-free if there exists some CFG $\mathcal{G}$ such that $L = \mathcal{L}(\mathcal{G})$.

Assorted remarks

- $X \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ is simply an abbreviation for a bunch of productions $X \to \alpha_1$, $X \to \alpha_2$, ..., $X \to \alpha_n$.
- These grammars are called context-free because a rule $X \to \alpha$ says that an X can *always* be expanded to α, no matter where the X occurs.
  This contrasts with context-sensitive rules, which might allow us to expand X only in certain contexts, e.g. $bXc \to b\alpha c$.
- Broad intuition: context-free languages allow nesting of structures to arbitrary depth. E.g. brackets, begin-end blocks, if-then-else statements, subordinate clauses in English, ...

Grammar for the Tiny Fragment of English

Grammar G1 generates the sentences on the previous slide:

Grammatical rules	**Lexical rules**
S → NP VP	Det → a | the | her (determiners)
NP → Det N	N → man | park | duck | telescope (nouns)
NP → Det N PP	Pro → you (pronoun)
NP → Pro	V → saw | walked | made (verbs)
VP → V NP PP	Prep → in | with | for (prepositions)
VP → V NP	
VP → V	
PP → Prep NP	

Does G1 produce a finite or an infinite number of sentences?

Recursion

Recursion in a grammar makes it possible to generate an infinite number of sentences.

In direct recursion, a non-terminal on the LHS of a rule also appears on its RHS. The following rules add direct recursion to G1:

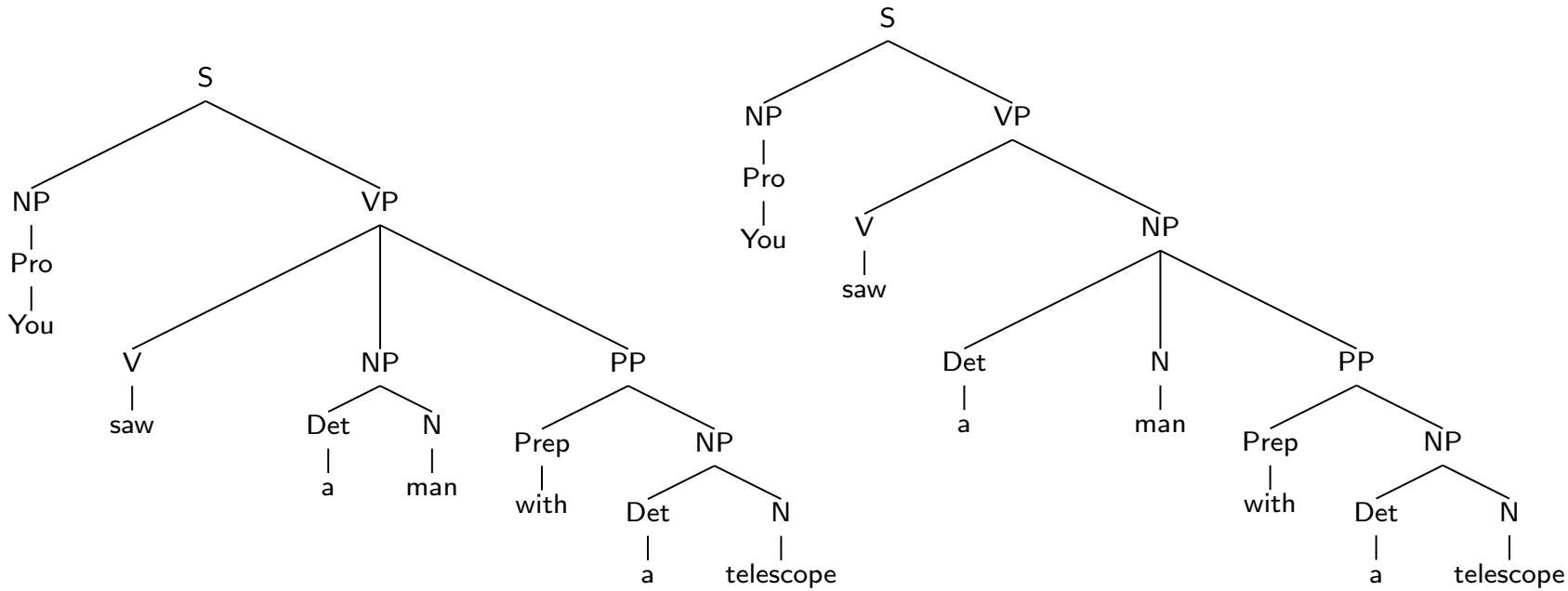
VP $\rightarrow$ VP Conj VP
Conj $\rightarrow$ and | or

In indirect recursion, some non-terminal can be expanded (via several steps) to a sequence of symbols containing that non-terminal:

NP $\rightarrow$ Det N PP
PP $\rightarrow$ Prep NP

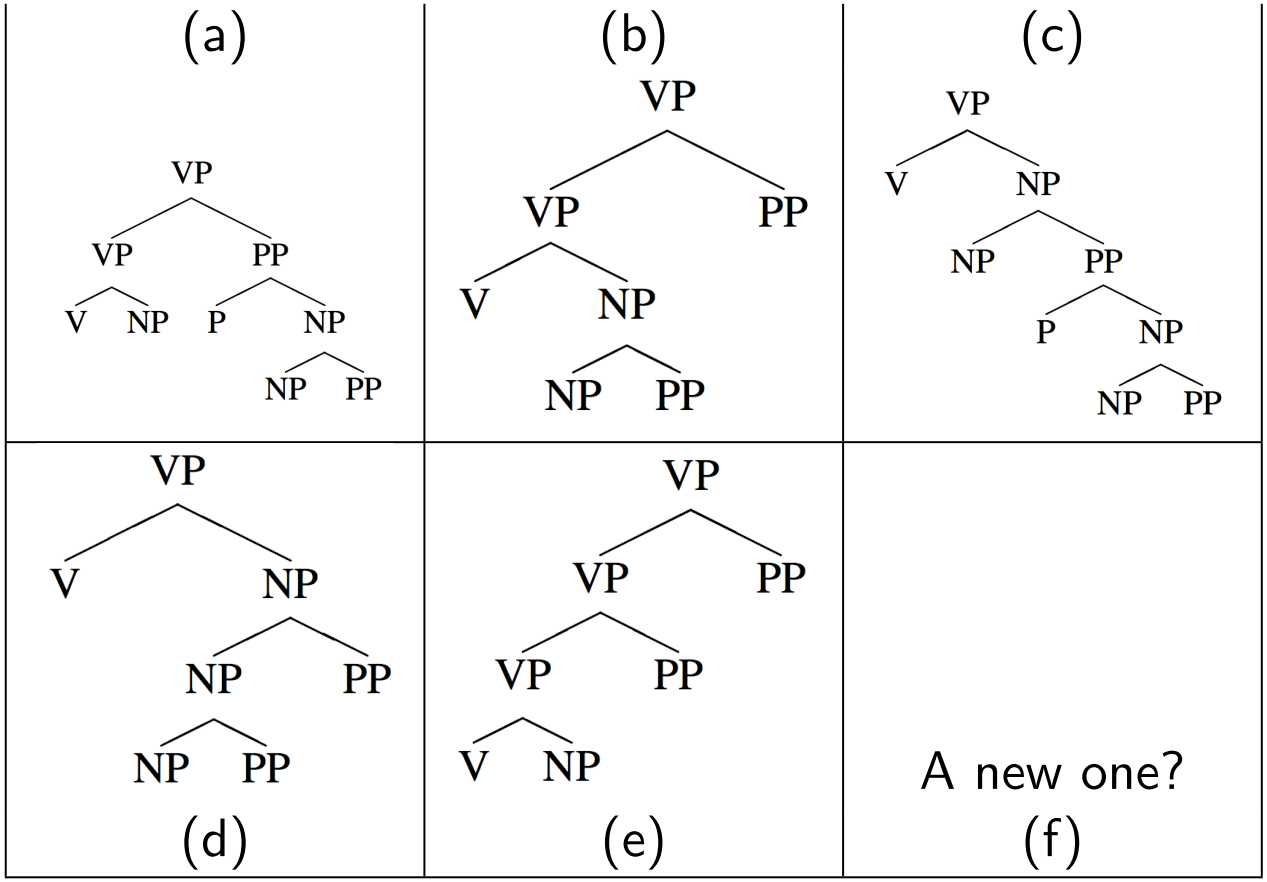
Structural Ambiguity

You saw a man with a telescope.



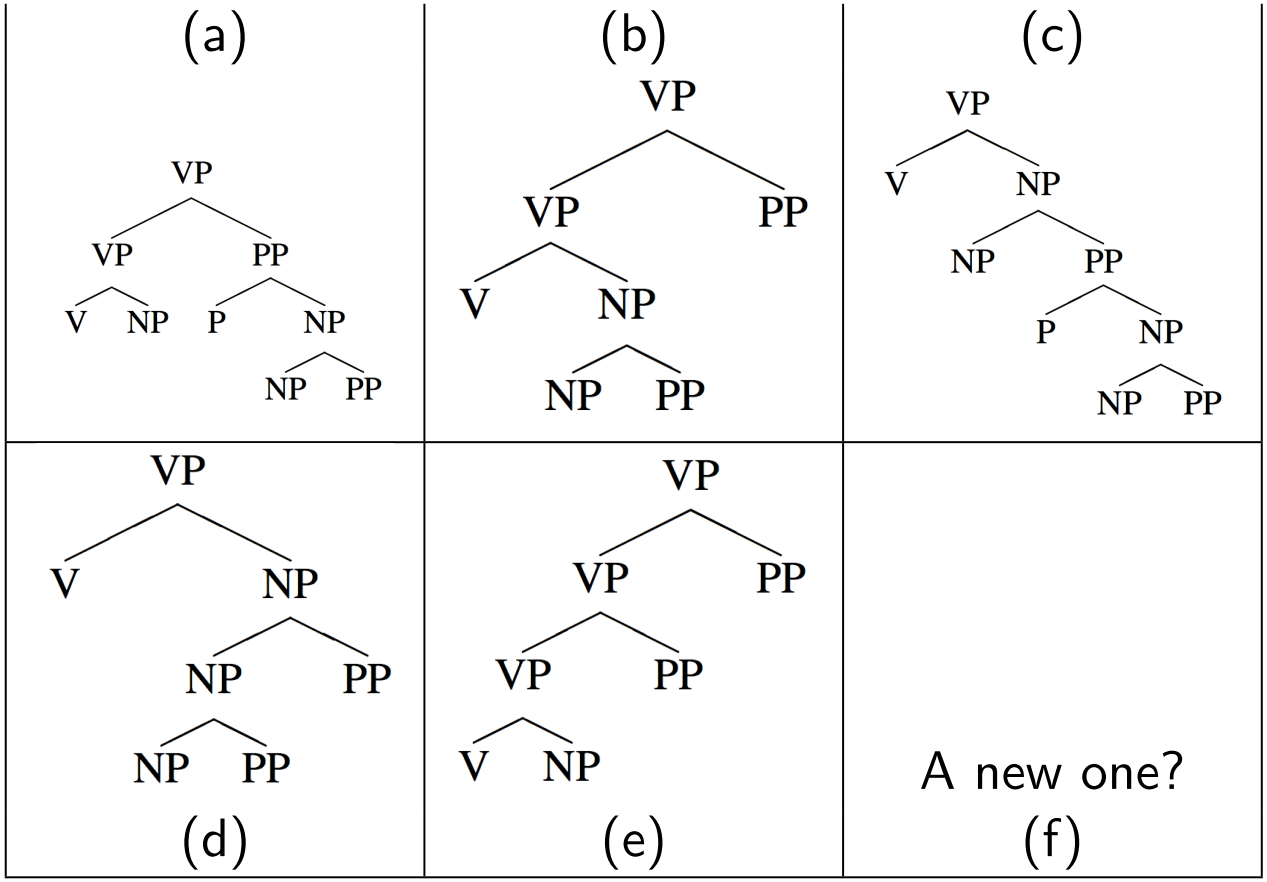
This illustrates attachment ambiguity: the PP can be a part of the VP or of the NP. Note that there's no POS ambiguity here.

A Fun Exercise - Which is the VP?



saw the car from my house window with my telescope

A Fun Exercise - Which is the VP?



saw the car from my house window with my telescope

E

Content

1 Grammars and syntax parsing

2 Parsing with context free grammars

3 Parsing with probabilistic context free grammars

4 Dependency parsing

5 Parsing with neural networks

Chomsky Normal Form

A context-free grammar is in Chomsky normal form if all productions are of the form $A \rightarrow B\ C$ or $A \rightarrow a$ where A, B, C are nonterminals in the grammar and a is a word in the grammar.

> *Disregarding the empty string, every CFG is equivalent to a grammar in Chomsky normal form (the grammars' string languages are identical)*

Why is that important?

- A normal form constrains the possible ways to represent an object
- Makes parsing efficient

Conversion to Chomsky Normal Form

- Replace all words in an RHS with a preterminal that rewrites to that word

- Break all RHSes into a sequence of RHSes with two nonterminals, possibly introducing new nonterminals:

$$S \rightarrow A_1 A_2 A_3$$

transforms into

$$S \rightarrow A_1 B$$

$$B \rightarrow A_2 A_3$$

.

Parsing algorithms

Goal: compute the structure(s) for an input string given a grammar.

- As usual, ambiguity is a huge problem.
  - For correctness: need to find the right structure to get the right meaning.
  - For efficiency: searching all possible structures can be very slow; want to use parsing for large-scale language tasks (e.g., used to create Google's "infoboxes").

Global and local ambiguity

- We've already seen examples of global ambiguity: multiple analyses for a full sentence, like I saw the man with the telescope
- But local ambiguity is also a big problem: multiple analyses for parts of sentence.
  - the dog bit the child: first three words could be NP (but aren't).
  - Building useless partial structures wastes time.
  - Avoiding useless computation is a major issue in parsing.
- Syntactic ambiguity is rampant; humans usually don't even notice because we are good at using context/semantics to disambiguate.

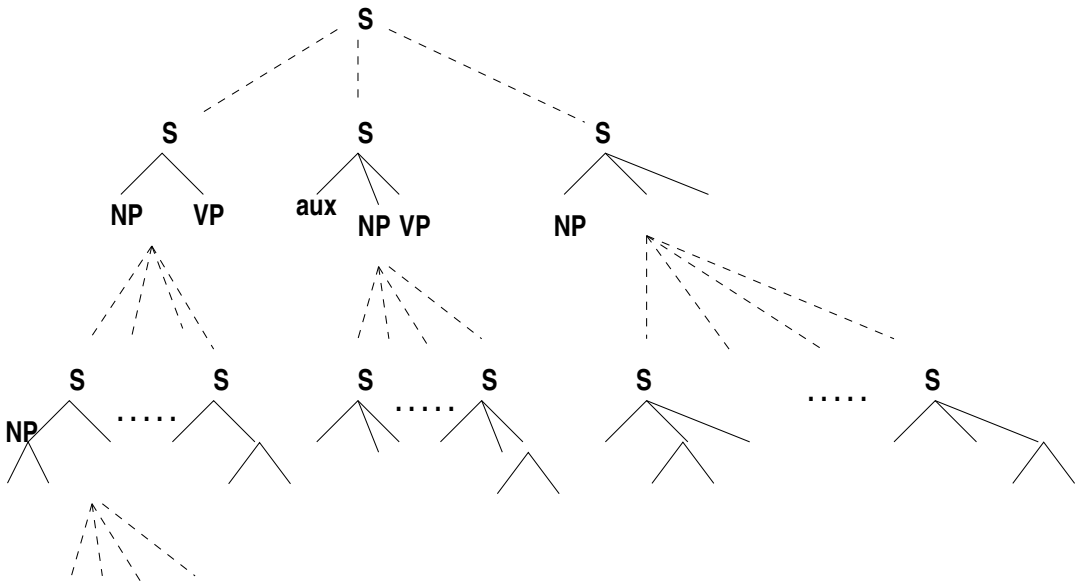
Parser properties

All parsers have two fundamental properties:

- Directionality: the sequence in which the structures are constructed.
  - top-down: start with root category (S), choose expansions, build down to words.
  - bottom-up: build subtrees over words, build up to S.
  - Mixed strategies also possible (e.g., left corner parsers)
- Search strategy: the order in which the search space of possible analyses is explored.

Example: search space for top-down parser

- Start with S node.
- Choose one of many possible expansions.
- Each of which has children with many possible expansions...
- etc



Shay Cohen, Accelerated Natural Language Processing (Slides)

Search strategies

- depth-first search: explore one branch of the search space at a time, as far as possible. If this branch is a dead-end, parser needs to backtrack.
- breadth-first search: expand all possible branches in parallel (or simulated parallel). Requires storing many incomplete parses in memory at once.
- best-first search: score each partial parse and pursue the highest-scoring options first. (Will get back to this when discussing statistical parsing.)

Shay Cohen, Accelerated Natural Language Processing (Slides)

Recursive Descent Parsing

- A recursive descent parser treats a grammar as a specification of how to break down a top-level goal (find `S`) into subgoals (find `NP VP`).
- It is a **top-down**, **depth-first** parser:
  - Blindly expand nonterminals until reaching a terminal (word).
  - If multiple options available, choose one but store current state as a backtrack point (in a **stack** to ensure depth-first.)
  - If terminal matches next input word, continue; else, backtrack.

RD Parsing algorithm

Start with subgoal = S, then repeat until input/subgoals are empty:

- If first subgoal in list is a **non-terminal** A, then pick an expansion A $\rightarrow$ B C from grammar and replace A in subgoal list with B C
- If first subgoal in list is a **terminal** w:
  - If input is empty, backtrack.
  - If next input word is different from w, backtrack.
  - If next input word is w, match! i.e., consume input word w and subgoal w and move to next subgoal.

If we run out of backtrack points but not input, no parse is possible.

Recursive descent example

Consider a very simple example:

- Grammar contains only these rules:

S → NP VP	VP → V	NN → bit	V → bit
NP → DT NN	DT → the	NN → dog	V → dog

- The input sequence is the dog bit

Recursive descent example

- Operations:
  - Expand (E)
  - Match (M)
  - Backtrack to step n (Bn)

Step	Op.	Subgoals	Input
0		S	the dog bit
1	E	NP VP	the dog bit
2	E	DT NN VP	the dog bit
3	E	the NN VP	the dog bit
4	M	NN VP	dog bit
5	E	bit VP	dog bit
6	B4	NN VP	dog bit
7	E	dog VP	dog bit
8	M	VP	bit
9	E	V	bit
10	E	bit	bit
11	M		

Shay Cohen, Accelerated Natural Language Processing (Slides)

Further notes

- The above sketch is actually a recognizer: it tells us whether the sentence has a valid parse, but not what the parse is. For a parser, we'd need more details to store the structure as it is built.
- We only had one backtrack, but in general things can be much worse!
  - If we have left-recursive rules like `NP` $\rightarrow$ `NP PP`, we get an infinite loop!

Shift-Reduce Parsing

A Shift-Reduce parser tries to find sequences of words and phrases that correspond to the righthand side of a grammar production and replace them with the lefthand side:

- **Directionality** = bottom-up: starts with the words of the input and tries to build trees from the words up.
- **Search strategy** = breadth-first: starts with the words, then applies rules with matching right hand sides, and so on until the whole sentence is reduced to an S.

Algorithm Sketch: Shift-Reduce Parsing

Until the words in the sentences are substituted with S:

- Scan through the input until we recognise something that corresponds to the RHS of one of the production rules (shift)
- Apply a production rule in reverse; i.e., replace the RHS of the rule which appears in the sentential form with the LHS of the rule (reduce)

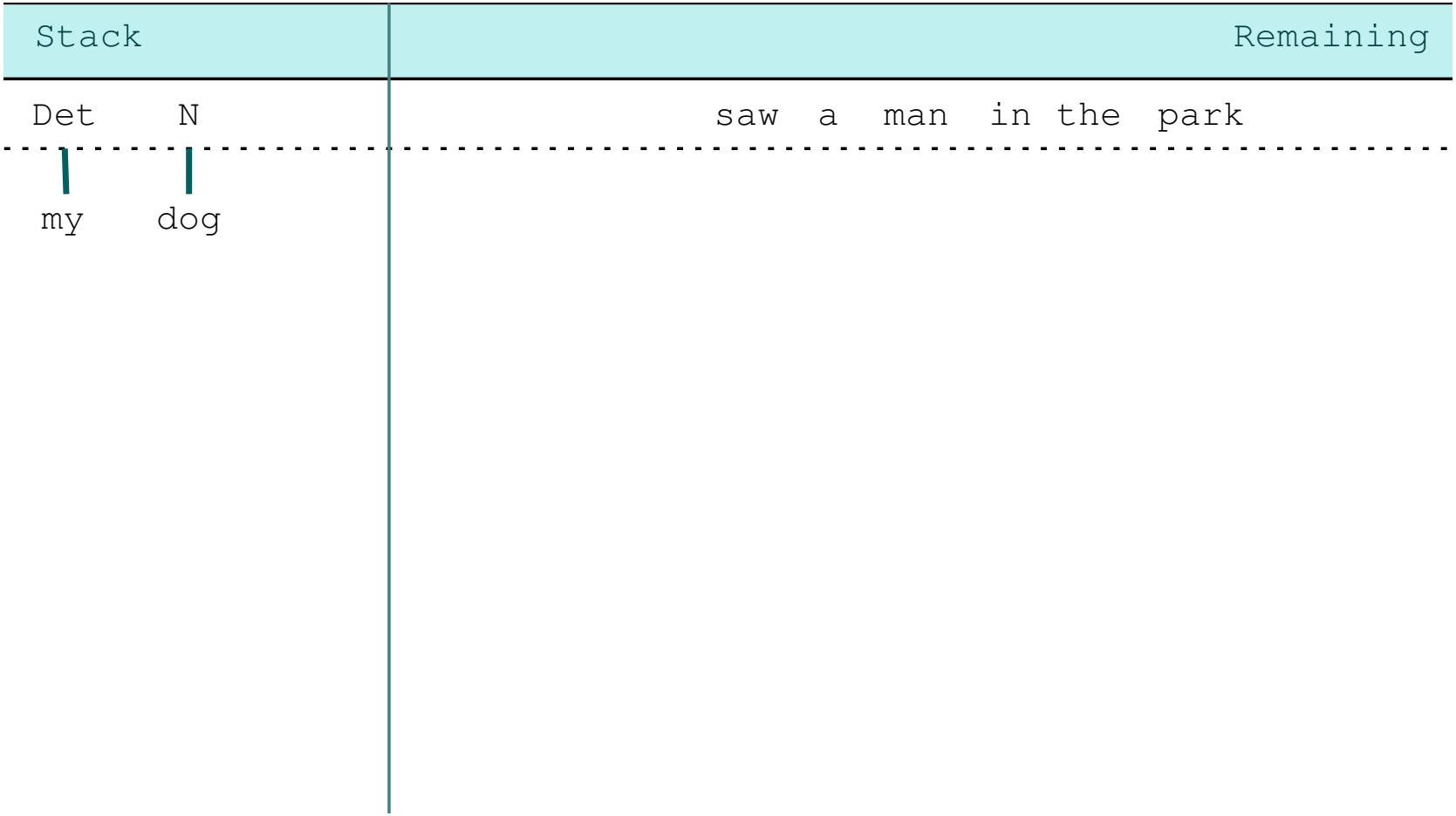
A shift-reduce parser implemented using a stack:

1. start with an empty stack
2. a shift action pushes the current input symbol onto the stack
3. a reduce action replaces n items with a single item

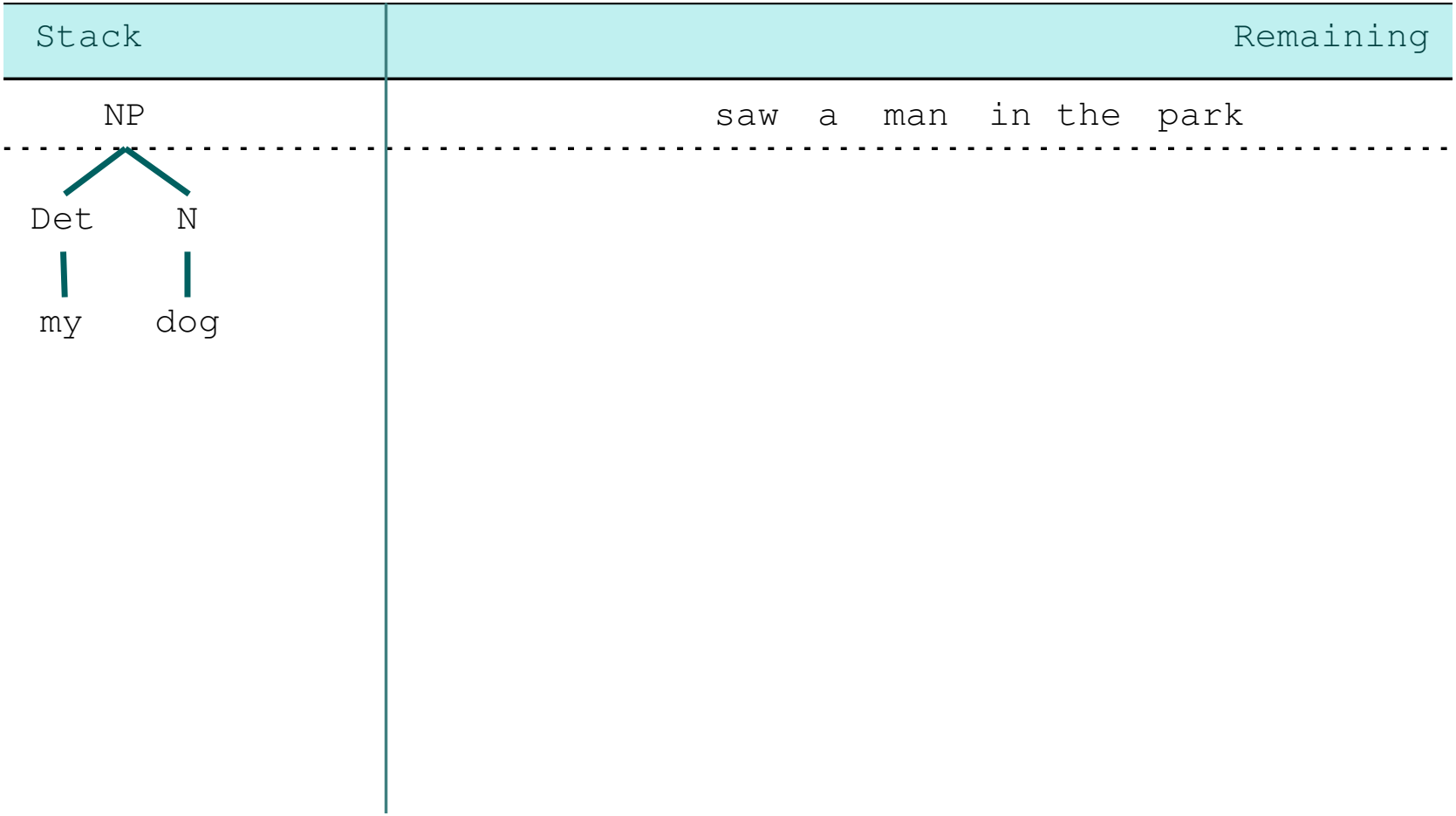
Shift-Reduce Parsing

Stack	Remaining
Det	dog saw a man in the park
my	

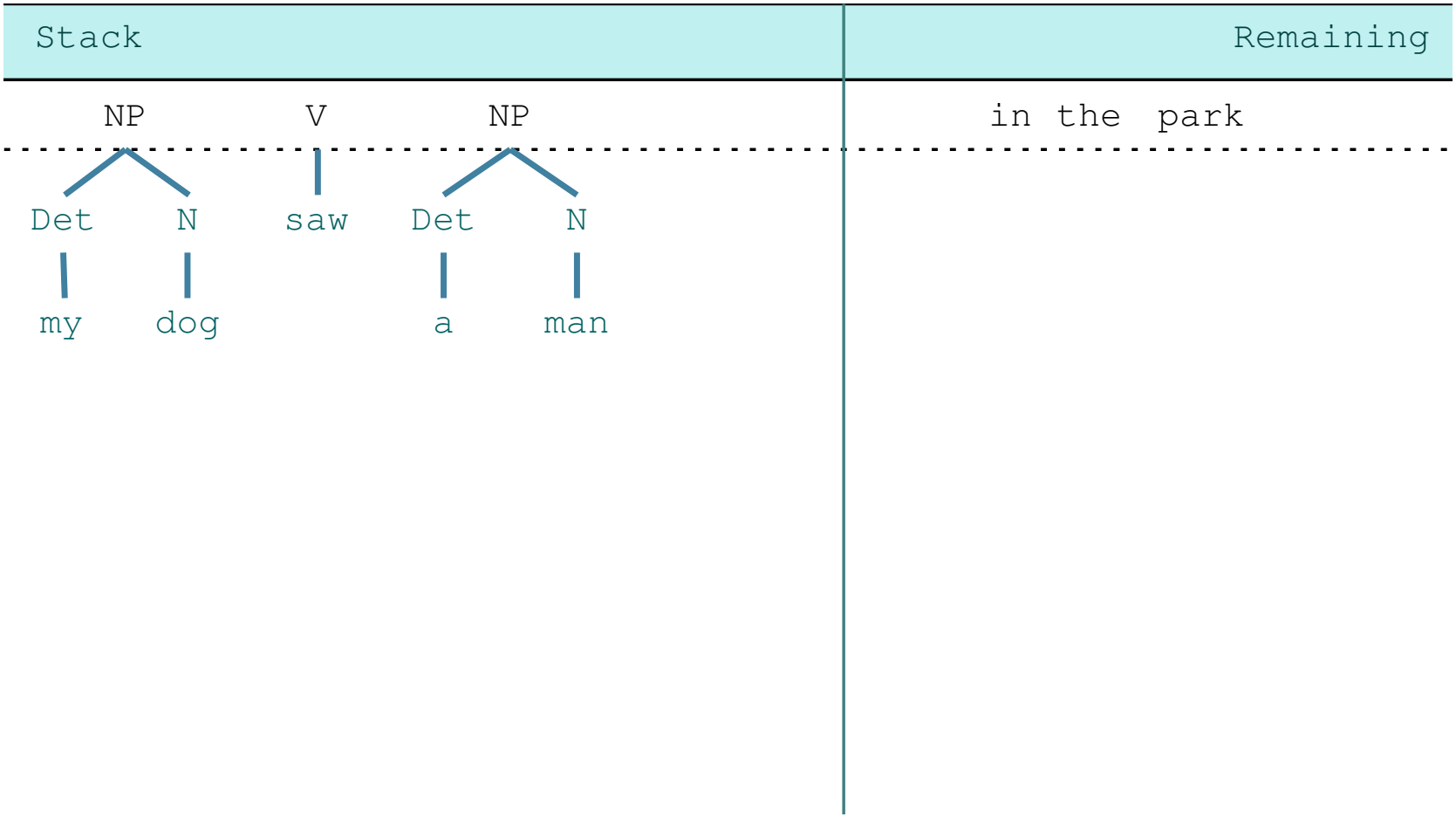
Shift-Reduce Parsing



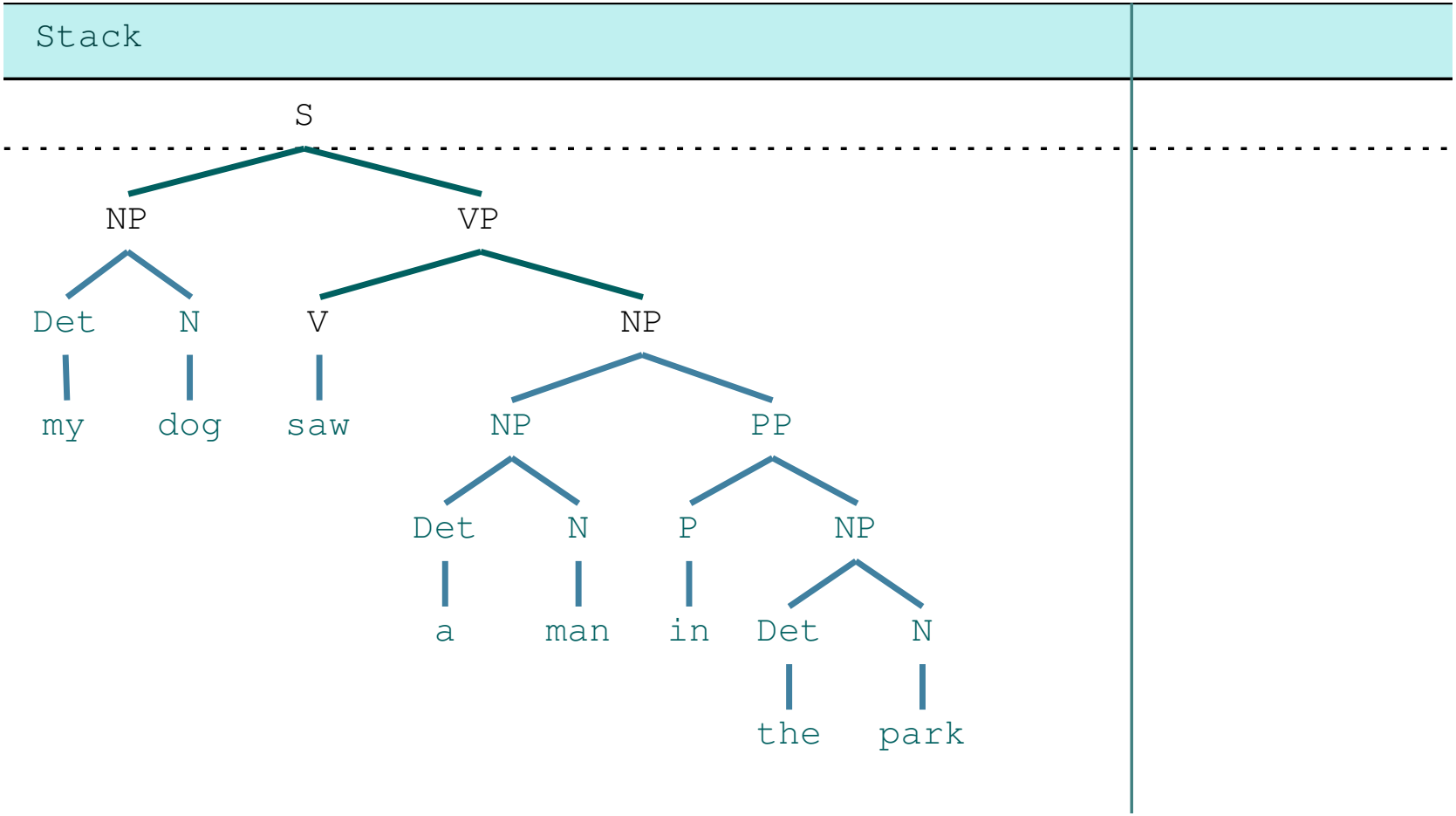
Shift-Reduce Parsing



Shift-Reduce Parsing

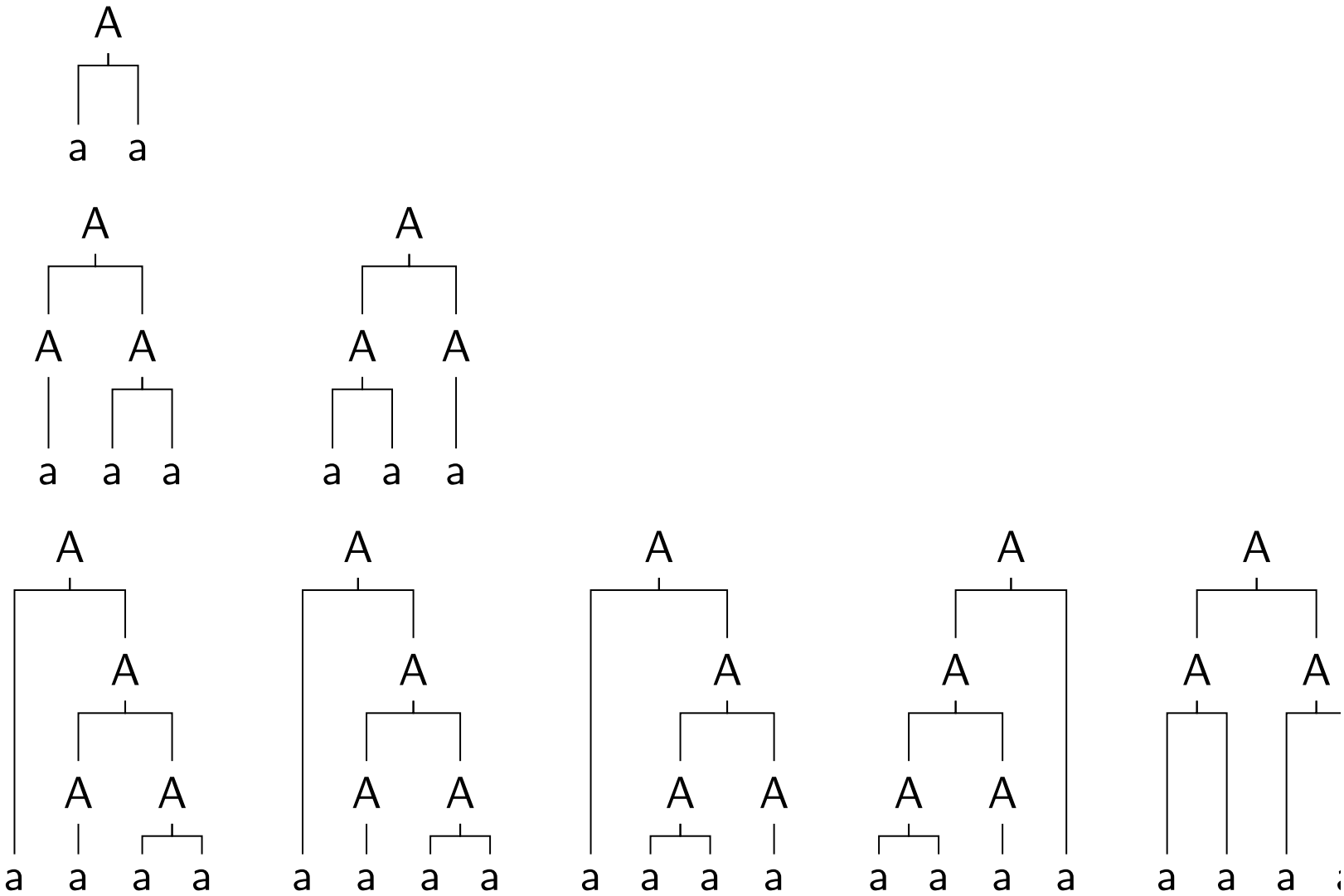


Shift-Reduce Parsing



Shay Cohen, Accelerated Natural Language Processing (Slides)

How many parses are there?



How many parses are there?

Intution. Let $C(n)$ be the number of binary trees over a sentence of length n. The root of this tree has two subtrees: one over k words ($1 \leq k < n$), and one over $n - k$ words. Hence, for all values of k, we can combine any subtree over k words with any subtree over $n - k$ words:

$$C(n) = \sum_{k=1}^{n-1} C(k) \times C(n-k)$$

$$C(n) = \frac{(2n)!}{(n+1)!n!}$$

These numbers are called the Catalan numbers. They're big numbers!

n	1	2	3	4	5	6	8	9	10	11	12
$C(n)$	1	1	2	5	14	42	132	429	1430	4862	16796

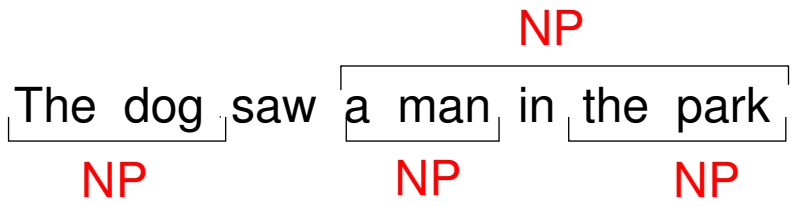
Problems with Parsing as Search

1. A **recursive descent parser** (top-down) will do badly if there are many different rules for the same LHS. Hopeless for rewriting parts of speech (preterminals) with words (terminals).
2. A **shift-reduce parser** (bottom-up) does a lot of useless work: many phrase structures will be locally possible, but globally impossible. Also inefficient when there is much lexical ambiguity.
3. Both strategies do repeated work by re-analyzing the same substring many times.

We will see how chart parsing solves the re-parsing problem, and also copes well with ambiguity.

Dynamic Programming

With a CFG, a parser should be able to avoid re-analyzing sub-strings because the analysis of any sub-string is independent of the rest of the parse.



The parser's exploration of its search space can exploit this independence if the parser uses dynamic programming.

Dynamic programming is the basis for all chart parsing algorithms.

Parsing as Dynamic Programming

- Given a problem, systematically fill a table of solutions to sub-problems: this is called memoization.
- Once solutions to all sub-problems have been accumulated, solve the overall problem by composing them.
- For parsing, the sub-problems are analyses of sub-strings and correspond to constituents that have been found.
- Sub-trees are stored in a chart (aka well-formed substring table), which is a record of all the substructures that have ever been built during the parse.

Solves **re-parsing problem**: sub-trees are looked up, not re-parsed!
Solves **ambiguity problem**: chart implicitly stores all parses!

Depicting a Chart

A chart can be depicted as a matrix:

- Rows and columns of the matrix correspond to the start and end positions of a span (ie, starting right before the first word, ending right after the final one);
- A cell in the matrix corresponds to the sub-string that starts at the row index and ends at the column index.
- It can contain information about the type of constituent (or constituents) that span(s) the substring, pointers to its sub-constituents, and/or predictions about what constituents might follow the substring.

CYK Algorithm

CYK (Cocke, Younger, Kasami) is an algorithm for recognizing and recording constituents in the chart.

- Assumes that the grammar is in Chomsky Normal Form: rules all have form $A \rightarrow BC$ or $A \rightarrow w$.
- Conversion to CNF can be done automatically.

NP	$\rightarrow$	Det Nom	NP	$\rightarrow$	Det Nom	
Nom	$\rightarrow$	N | OptAP Nom	Nom	$\rightarrow$	*book* | *orange* | AP Nom	
OptAP	$\rightarrow$	ϵ | OptAdv A	AP	$\rightarrow$	*heavy* | *orange* | Adv A	
A	$\rightarrow$	*heavy* | *orange*	A	$\rightarrow$	*heavy* | *orange*	
Det	$\rightarrow$	*a*	Det	$\rightarrow$	*a*	
OptAdv	$\rightarrow$	ϵ | *very*	Adv	$\rightarrow$	*very*	
N	$\rightarrow$	*book* | *orange*				

CYK: an example

Let's look at a simple example before we explain the general case.

Grammar Rules in CNF

NP	$\rightarrow$	Det Nom
Nom	$\rightarrow$	*book* | *orange* | AP Nom
AP	$\rightarrow$	*heavy* | *orange* | Adv A
A	$\rightarrow$	*heavy* | *orange*
Det	$\rightarrow$	*a*
Adv	$\rightarrow$	*very*

(N.B. Converting to CNF sometimes breeds duplication!)

Now let's parse: *a very heavy orange book*

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a					
1	very					
2	heavy					
3	orange					
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det				
1	very					
2	heavy					
3	orange					
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det				
1	very		Adv			
2	heavy					
3	orange					
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det				
1	very		Adv			
2	heavy			A,AP		
3	orange					
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det				
1	very		Adv	AP		
2	heavy			A,AP		
3	orange					
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det				
1	very		Adv	AP		
2	heavy			A,AP		
3	orange				Nom,A,AP	
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det				
1	very		Adv	AP		
2	heavy			A,AP	Nom	
3	orange				Nom,A,AP	
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det				
1	very		Adv	AP	Nom	
2	heavy			A,AP	Nom	
3	orange				Nom,A,AP	
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	
1	very		Adv	AP	Nom	
2	heavy			A,AP	Nom	
3	orange				Nom,A,AP	
4	book					

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	
1	very		Adv	AP	Nom	
2	heavy			A,AP	Nom	
3	orange				Nom,A,AP	
4	book					Nom

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	
1	very		Adv	AP	Nom	
2	heavy			A,AP	Nom	
3	orange				Nom,A,AP	Nom
4	book					Nom

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	
1	very		Adv	AP	Nom	
2	heavy			A,AP	Nom	Nom
3	orange				Nom,A,AP	Nom
4	book					Nom

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	
1	very		Adv	AP	Nom	Nom
2	heavy			A,AP	Nom	Nom
3	orange				Nom,A,AP	Nom
4	book					Nom

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	NP
1	very		Adv	AP	Nom	Nom
2	heavy			A,AP	Nom	Nom
3	orange				Nom,A,AP	Nom
4	book					Nom

CYK: The general algorithm

function CKY-Parse(*words*, *grammar*) **returns** *table* **for**

$j \leftarrow$**from** 1 **to** LENGTH(*words*) **do**
  $table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$
  for $i \leftarrow$**from** $j-2$ **downto** 0 **do**
    for $k \leftarrow i+1$ **to** $j-1$ **do**
      $table[i,j] \leftarrow table[i,j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i,k]$
          $C \in table[k,j]\}$

CYK: The general algorithm

function CKY-Parse(*words*, *grammar*) **returns** *table* **for**

$j \leftarrow$**from** 1 **to** LENGTH(*words*) **do** — loop over the columns

  $table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$ — fill bottom cell

  for $i \leftarrow$**from** $j-2$ **downto** 0 **do** — fill row i in column j

    for $k \leftarrow i+1$ **to** $j-1$ **do** — loop over split locations between i and j

      $table[i, j] \leftarrow table[i, j] \cup$

$$\{A \mid A \rightarrow BC \in grammar, \; B \in table[i, k], \; C \in table[k, j]\}$$

Check the grammar for rules that link the constituent in $[i, k]$ with those in $[k, j]$. For each rule found store LHS in cell $[i, j]$.

A succinct representation of CKY

We have a Boolean table called Chart, such that $\mathrm{Chart}[A, i, j]$ is true if there is a sub-phrase according the grammar that dominates words i through words j

Build this chart recursively, similarly to the Viterbi algorithm:

For $j > i + 1$:

$$\mathrm{Chart}[A, i, j] = \bigvee_{k=i+1}^{j-1} \bigvee_{A \to B\ C} \mathit{Chart}[B, i, k] \wedge \mathit{Chart}[C, k, j]$$

Seed the chart, for $i + 1 = j$:
$\mathrm{Chart}[A, i, i + 1] = \mathrm{True}$ if there exists a rule $A \to w_{i+1}$ where w_{i+1} is the $(i + 1)$th word in the string

From CYK Recognizer to CYK Parser

- So far, we just have a chart recognizer, a way of determining whether a string belongs to the given language.
- Changing this to a parser requires recording which existing constituents were combined to make each new constituent.
- This requires another field to record the one or more ways in which a constituent spanning (i,j) can be made from constituents spanning (i,k) and (k,j). (More clearly displayed in graph representation, see next lecture.)
- In any case, for a fixed grammar, the CYK algorithm runs in time $O(n^3)$ on an input string of n tokens.
- The algorithm identifies all possible parses.

CYK-style parse charts

Even without converting a grammar to CNF, we can draw *CYK-style* parse charts:

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	NP
1	very		OptAdv	OptAP	Nom	Nom
2	heavy			A,OptAP	Nom	Nom
3	orange				N,Nom,A,AP	Nom
4	book					N,Nom

(We haven't attempted to show ϵ-phrases here. Could in principle use cells below the main diagonal for this . . .)
However, CYK-style parsing will have run-time worse than $O(n^3)$ if e.g. the grammar has rules $A \rightarrow BCD$.

Dynamic Programming as a problem-solving technique

- Given a problem, systematically fill a table of solutions to sub-problems: this is called memoization.
- Once solutions to all sub-problems have been accumulated, solve the overall problem by composing them.
- For parsing, the sub-problems are analyses of sub-strings and correspond to constituents that have been found.
- Sub-trees are stored in a chart (aka well-formed substring table), which is a record of all the substructures that have ever been built during the parse.

Solves **re-parsing problem**: sub-trees are looked up, not re-parsed!
Solves **ambiguity problem**: chart implicitly stores all parses!

Shay Cohen, Accelerated Natural Language Processing (Slides)

Content

1 Grammars and syntax parsing

2 Parsing with context free grammars

3 Parsing with probabilistic context free grammars

4 Dependency parsing

5 Parsing with neural networks

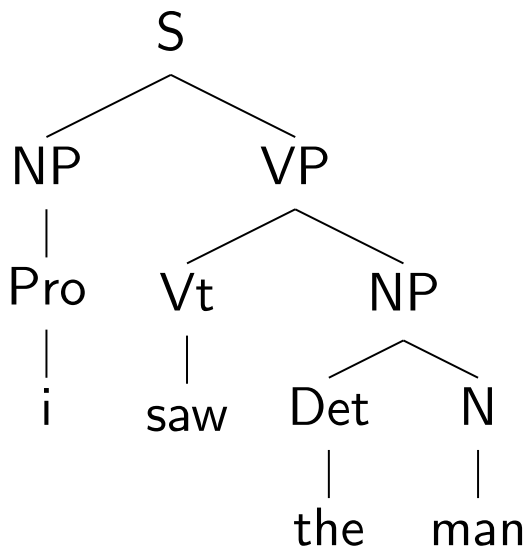
Treebank grammars

- The big idea: instead of paying linguists to write a grammar, pay them to annotate real sentences with parse trees.
- This way, we implicitly get a grammar (for CFG: read the rules off the trees).
- **And** we get probabilities for those rules (using any of our favorite estimation techniques).
- We can use these probabilities to improve disambiguation and even speed up parsing.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Treebank grammars

For example, if we have this tree in our corpus:



Then we add rules

$\texttt{S} \rightarrow \texttt{NP VP}$

$\texttt{NP} \rightarrow \texttt{Pro}$

$\texttt{Pro} \rightarrow \texttt{i}$

$\texttt{VP} \rightarrow \texttt{Vt NP}$

$\texttt{Vt} \rightarrow \texttt{saw}$

$\texttt{NP} \rightarrow \texttt{Det N}$

$\texttt{Art} \rightarrow \texttt{the}$

$\texttt{N} \rightarrow \texttt{man}$

With more trees, we can start to count rules and estimate their probabilities.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Example: The Penn Treebank

- The first large-scale parse annotation project, begun in 1989.
- Original corpus of syntactic parses: Wall Street Journal text
  - About 40,000 annotated sentences (1m words)
  - Standard phrasal categories like `S`, `NP`, `VP`, `PP`.
- Now many other data sets (e.g. transcribed speech), and different kinds of annotation; also inspired treebanks in many other languages.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Other language treebanks

- Many annotated with **dependency grammar** rather than CFG (see next lecture).
- Some require paid licenses, others are free.
- Just a few examples:
  - Danish Dependency Treebank
  - Alpino Treebank (Dutch)
  - Bosque Treebank (Portuguese)
  - Talbanken (Swedish)
  - Prague Dependency Treebank (Czech)
  - TIGER corpus, Tuebingen Treebank, NEGRA corpus (German)
  - Penn Chinese Treebank
  - Penn Arabic Treebank
  - Tuebingen Treebank of Spoken Japanese, Kyoto Text Corpus

Shay Cohen, Accelerated Natural Language Processing (Slides)

Creating a treebank PCFG

A **probabilistic context-free grammar** (PCFG) is a CFG where each rule $\mathtt{A} \rightarrow \alpha$ (where α is a symbol sequence) is assigned a probability $P(\alpha|A)$.

- The sum over all expansions of $\mathtt{A}$ must equal 1: $\sum_{\alpha'} P(\alpha'|A) = 1$.
- Easiest way to create a PCFG from a treebank: MLE
  - Count all occurrences of $\mathtt{A} \rightarrow \alpha$ in treebank.
  - Divide by the count of all rules whose LHS is $\mathtt{A}$ to get $P(\alpha|A)$
- But as usual many rules have very low frequencies, so MLE isn't good enough and we need to smooth.

Shay Cohen, Accelerated Natural Language Processing (Slides)

The generative model

Like *n*-gram models and HMMs, PCFGs are a **generative model**. Assumes sentences are generated as follows:

- Start with root category S.
- Choose an expansion α for S with probability $P(\alpha|S)$.
- Then recurse on each symbol in α.
- Continue until all symbols are terminals (nothing left to expand).

Shay Cohen, Accelerated Natural Language Processing (Slides)

The probability of a parse

- Under this model, the probability of a parse t is simply the product of all rules in the parse:

$$P(t) = \prod_{A \to \alpha \in t} p(A \to \alpha \mid A)$$

Shay Cohen, Accelerated Natural Language Processing (Slides)

Statistical disambiguation example

How can parse probabilities help disambiguate PP attachment?

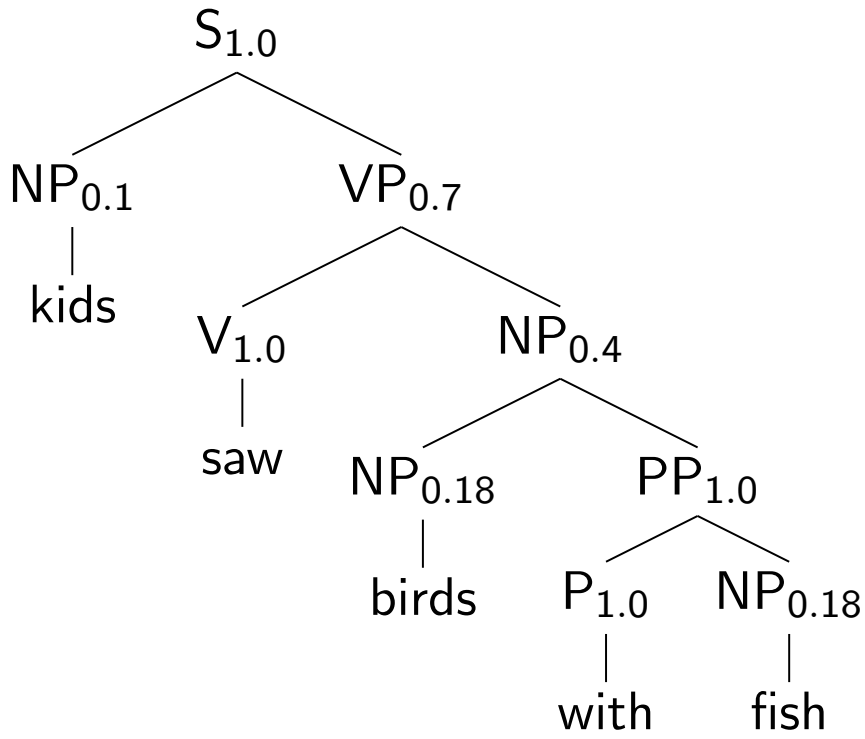
- Let's use the following PCFG, inspired by Manning & Schuetze (1999):

$S \rightarrow NP\ VP$	1.0	$NP \rightarrow NP\ PP$	0.4
$PP \rightarrow P\ NP$	1.0	$NP \rightarrow kids$	0.1
$VP \rightarrow V\ NP$	0.7	$NP \rightarrow birds$	0.18
$VP \rightarrow VP\ PP$	0.3	$NP \rightarrow saw$	0.04
$P \rightarrow with$	1.0	$NP \rightarrow fish$	0.18
$V \rightarrow saw$	1.0	$NP \rightarrow binoculars$	0.1

- We want to parse kids saw birds with fish.

Shay Cohen, Accelerated Natural Language Processing (Slides)

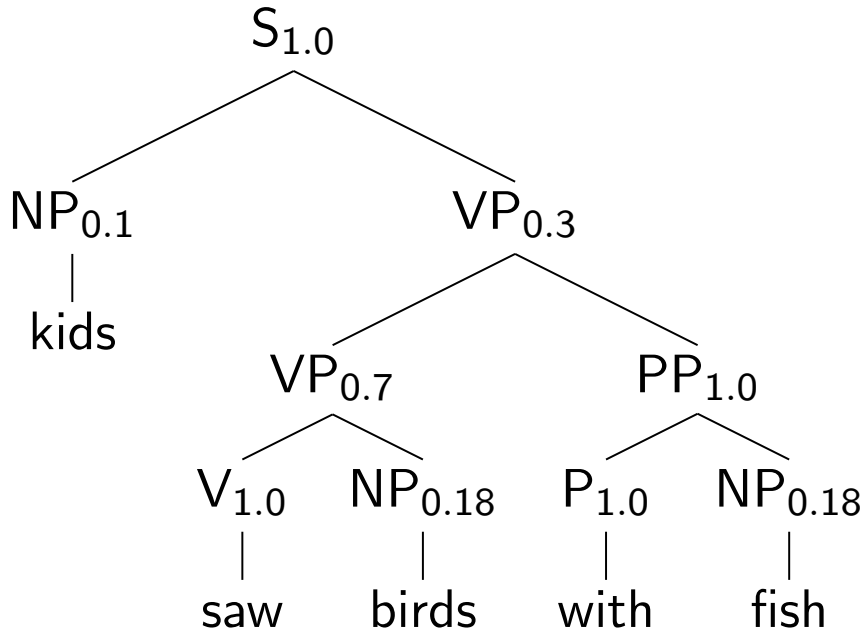
Probability of parse 1



- $P(t_1) = 1.0 \cdot 0.1 \cdot 0.7 \cdot 1.0 \cdot 0.4 \cdot 0.18 \cdot 1.0 \cdot 1.0 \cdot 0.18 = 0.0009072$

Shay Cohen, Accelerated Natural Language Processing (Slides)

Probability of parse 2



- $P(t_2) = 1.0 \cdot 0.1 \cdot 0.3 \cdot 0.7 \cdot 1.0 \cdot 0.18 \cdot 1.0 \cdot 1.0 \cdot 0.18 = 0.0006804$
- which is less than $P(t_1) = 0.0009072$, so t_1 is preferred. Yay!

Shay Cohen, Accelerated Natural Language Processing (Slides)

The probability of a sentence

- Since t implicitly includes the words $\vec{w}$, we have $P(t) = P(t, \vec{w})$.
- So, we also have a **language model**. Sentence probability is obtained by summing over $T(\vec{w})$, the set of valid parses of $\vec{w}$:

$$P(\vec{w}) = \sum_{t \in T(\vec{w})} P(t, \vec{w}) = \sum_{t \in T(\vec{w})} P(t)$$

- In our example, $P(\text{kids saw birds with fish}) = 0.0006804 + 0.0009072$.

Shay Cohen, Accelerated Natural Language Processing (Slides)

How to find the best parse?

First, remember standard CKY algorithm.

- Fills in cells in well-formed substring table (chart) by combining previously computed child cells.

	1	2	3	4
0	Pro, NP			S
1		Vt,Vp,N		VP
2			Pro, PosPro, D	NP
3				N,Vi
	$_0$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

Probabilistic CKY

It is straightforward to extend CKY parsing to the probabilistic case.

- Goal: return the highest probability parse of the sentence.
  - When we find an A spanning (i, j), store its probability along with its label and backpointers in cell (i, j)
  - If we later find an A with the same span but higher probability, replace the probability for A in cell (i, j), and update the backpointers to the new children.
- Analogous to Viterbi: we iterate over all possible child pairs (rather than previous states) and store the probability and backpointers for the best one.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Probabilistic CKY

We also have analogues to the other HMM algorithms.

- The **inside algorithm** computes the probability of the sentence (analogous to forward algorithm)
  - Same as above, but instead of storing the *best* parse for `A`, store the *sum* of all parses.
- The **inside-outside algorithm** algorithm is a form of EM that learns grammar rule probs from unannotated sentences (analogous to forward-backward).

Shay Cohen, Accelerated Natural Language Processing (Slides)

Best-first probabilistic parsing

- So far, we've been assuming **exhaustive** parsing: return all possible parses.
- But treebank grammars are huge!!
  - Exhaustive parsing of WSJ sentences up to 40 words long adds on average over 1m items to chart per sentence.[1]
  - Can be hundreds of possible parses, but most have extremely low probability: do we really care about finding these?
- **Best-first** parsing can help.

[1] Charniak, Goldwater, and Johnson, WVLC 1998.

Best-first probabilistic parsing

Use probabilities of subtrees to decide which ones to build up further.

- Each time we find a new constituent, we give it a **score** ("figure of merit") and add it to an **agenda**[2], which is ordered by score.
- Then we pop the next item off the agenda, add it to the chart, and see which new constituents we can make using it.
- We add those to the agenda, and iterate.

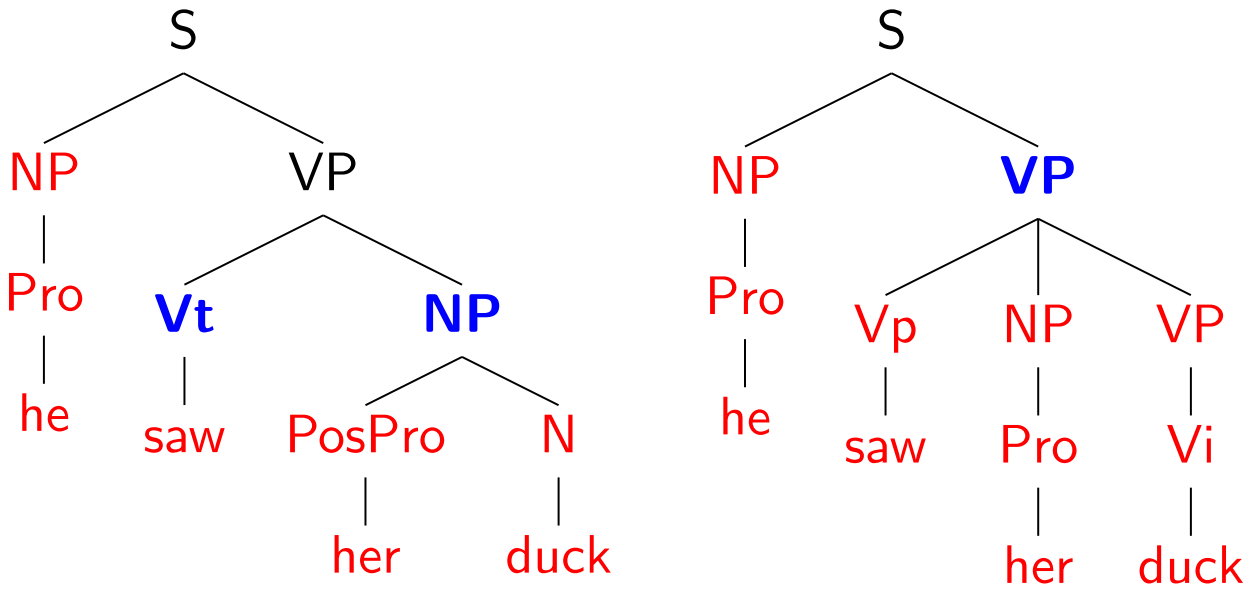
Notice we are no longer filling the chart in any fixed order.
Many variations on this idea, often limiting the size of the agenda by **pruning** out low-scoring edges (**beam search**).

[2] aka a priority queue

Shay Cohen, Accelerated Natural Language Processing (Slides)

Best-first intuition

Suppose red constituents are in chart already; **blue** are on agenda.

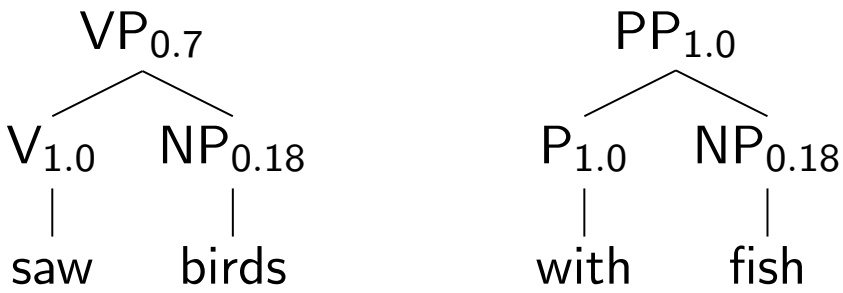


If the VP in right-hand tree scores high enough, we'll pop that next, add it to chart, then find the S. So, we could complete the whole parse before even finding the alternative VP.

Shay Cohen, Accelerated Natural Language Processing (Slides)

How do we score constituents?

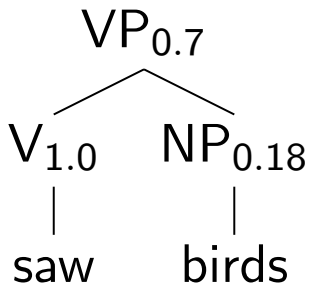
Perhaps according to the probability of the subtree they span? So, P(left example)=(0.7)(0.18) and P(right example)=0.18.

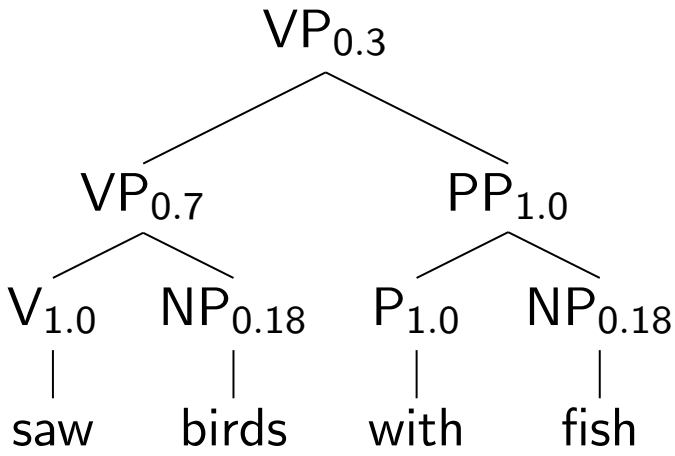


Shay Cohen, Accelerated Natural Language Processing (Slides)

How do we score constituents?

But what about comparing different sized constituents?





Shay Cohen, Accelerated Natural Language Processing (Slides)

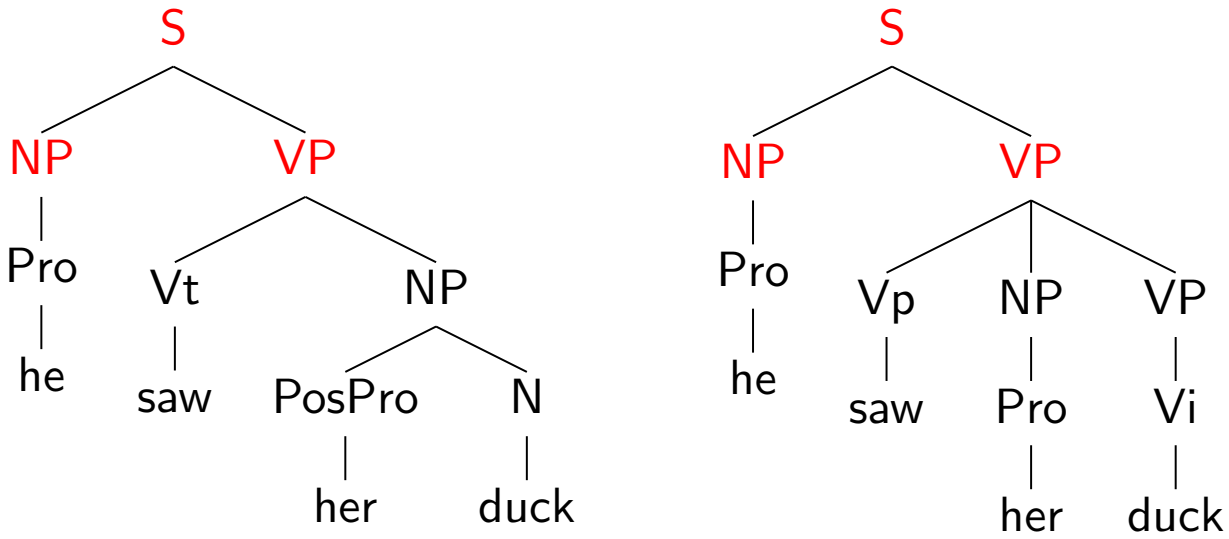
A better figure of merit

- If we use raw probabilities for the score, **smaller** constituents will almost always have higher scores.
  - Meaning we pop all the small constituents off the agenda before the larger ones.
  - Which would be very much like exhaustive bottom-up parsing!
- Instead, we can divide by the **number of words** in the constituent.
  - Very much like we did when comparing language models (recall **per-word** cross-entropy)!
- This works much better, though still not guaranteed to find the best parse first. Other improvements are possible.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Evaluating parse accuracy

Compare **gold standard** tree (left) to **parser output** (right):

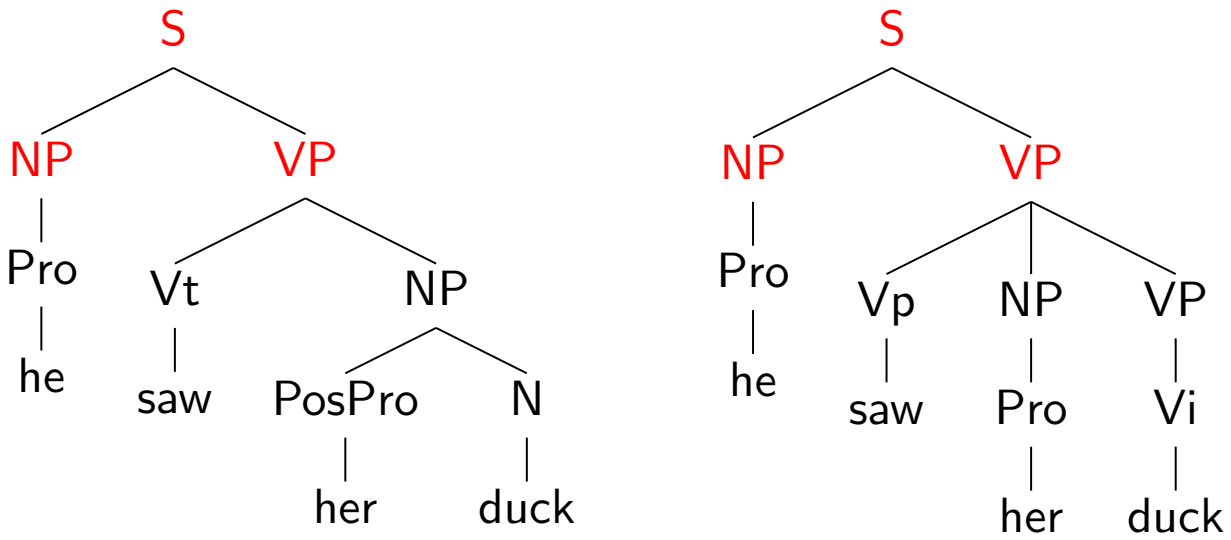


- Output constituent is counted correct if there is a gold constituent that spans the same sentence positions.
- Harsher measure: also require the constituent labels to match.
- Pre-terminals don't count as constituents.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Evaluating parse accuracy

Compare **gold standard** tree (left) to **parser output** (right):



- **Precision**: (# correct constituents)/(# in parser output) = 3/5
- **Recall**: (# correct constituents)/(# in gold standard) = 3/4
- **F-score**: balances precision/recall: 2pr/(p+r)

Shay Cohen, Accelerated Natural Language Processing (Slides)

Content

1 Grammars and syntax parsing

2 Parsing with context free grammars

3 Parsing with probabilistic context free grammars

4 Dependency parsing

5 Parsing with neural networks

Parsing: where are we now?

Parsing is not just WSJ. Lots of situations are much harder!

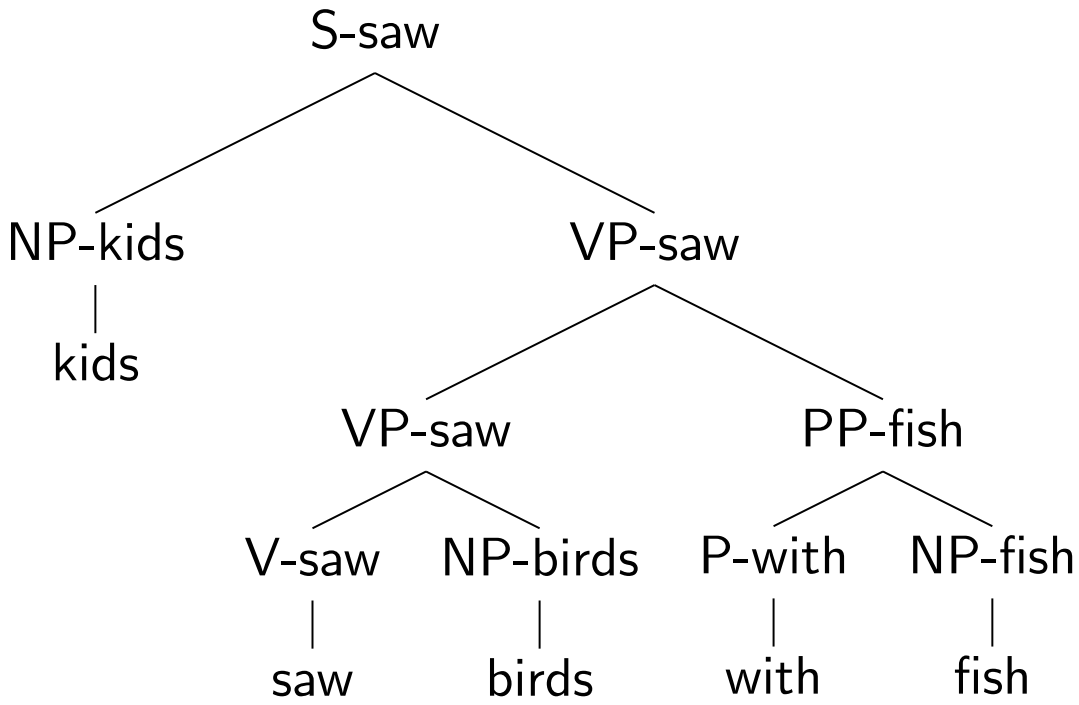
- Other languages, esp with free word order (up next) or little annotated data.
- Other domains, esp with jargon (e.g., biomedical) or non-standard language (e.g., social media text).

In fact, due to increasing focus on multilingual NLP, constituency syntax/parsing (English-centric) is losing ground to **dependency parsing**...

Shay Cohen, Accelerated Natural Language Processing (Slides)

Lexicalization, again

We saw that adding **lexical head** of the phrase can help choose the right parse:

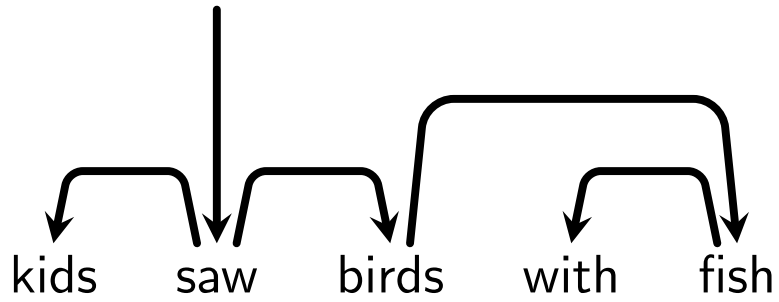


Dependency syntax focuses on the head-dependent relationships.

Dependency syntax

An alternative approach to sentence structure.

- A fully lexicalized formalism: no phrasal categories.
- Assumes *binary, asymmetric* grammatical relations between words: **head-dependent** relations, shown as directed edges:

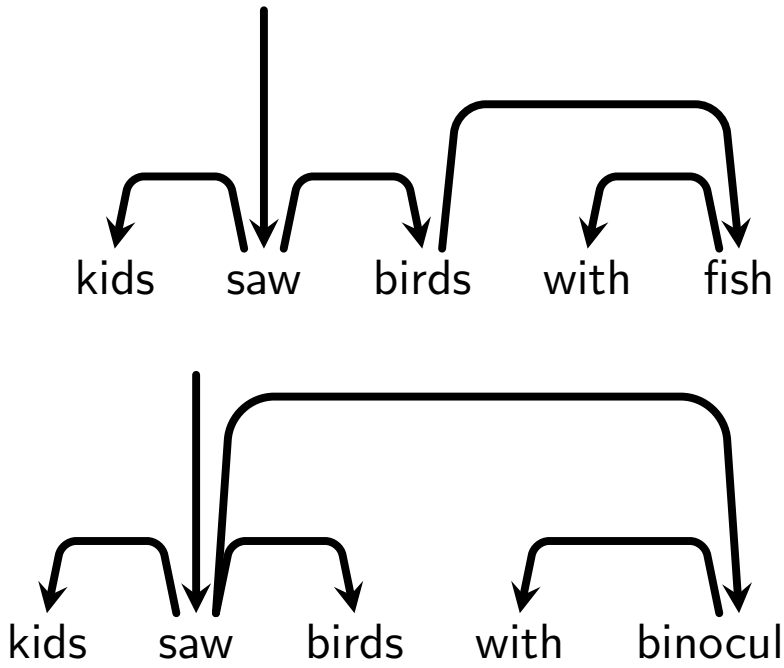


- Here, edges point from heads to their dependents.

Dependency trees

A valid dependency tree for a sentence requires:

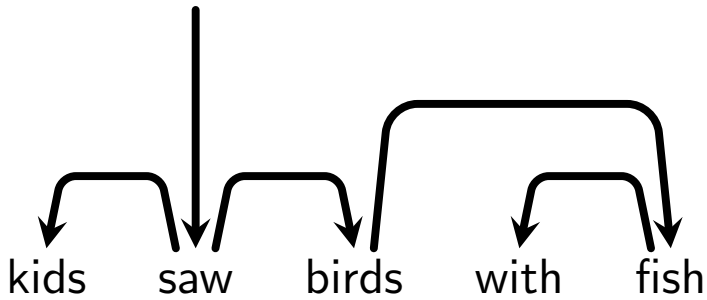
- A single distinguished **root** word.
- All other words have exactly one incoming edge.
- A unique path from the root to each other word.

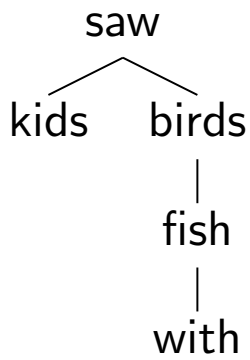


Shay Cohen, Accelerated Natural Language Processing (Slides)

It really is a tree!

- The usual way to show dependency trees is with edges over ordered sentences.
- But the edge structure (without word order) can also be shown as a more obvious tree:

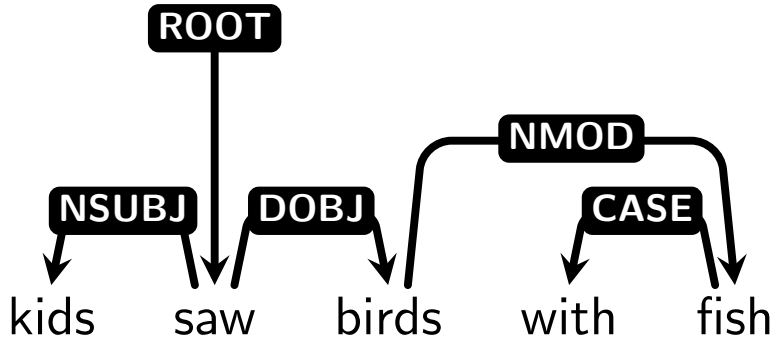




Shay Cohen, Accelerated Natural Language Processing (Slides)

Labelled dependencies

It is often useful to distinguish different kinds of head → modifier relations, by labelling edges:

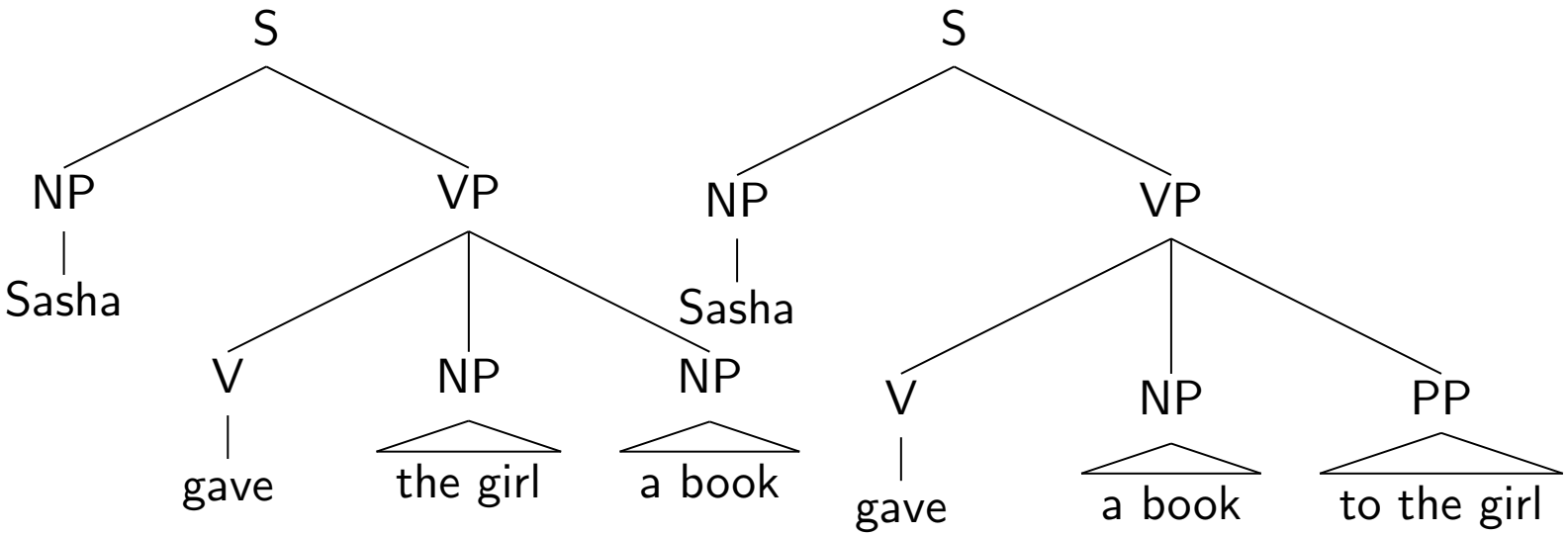


- Historically, different treebanks/languages used different labels.
- Now, the **Universal Dependencies** project aims to standardize labels and annotation conventions, bringing together annotated corpora from over 50 languages.
- Labels in this example (and in textbook) are from UD.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Why dependencies??

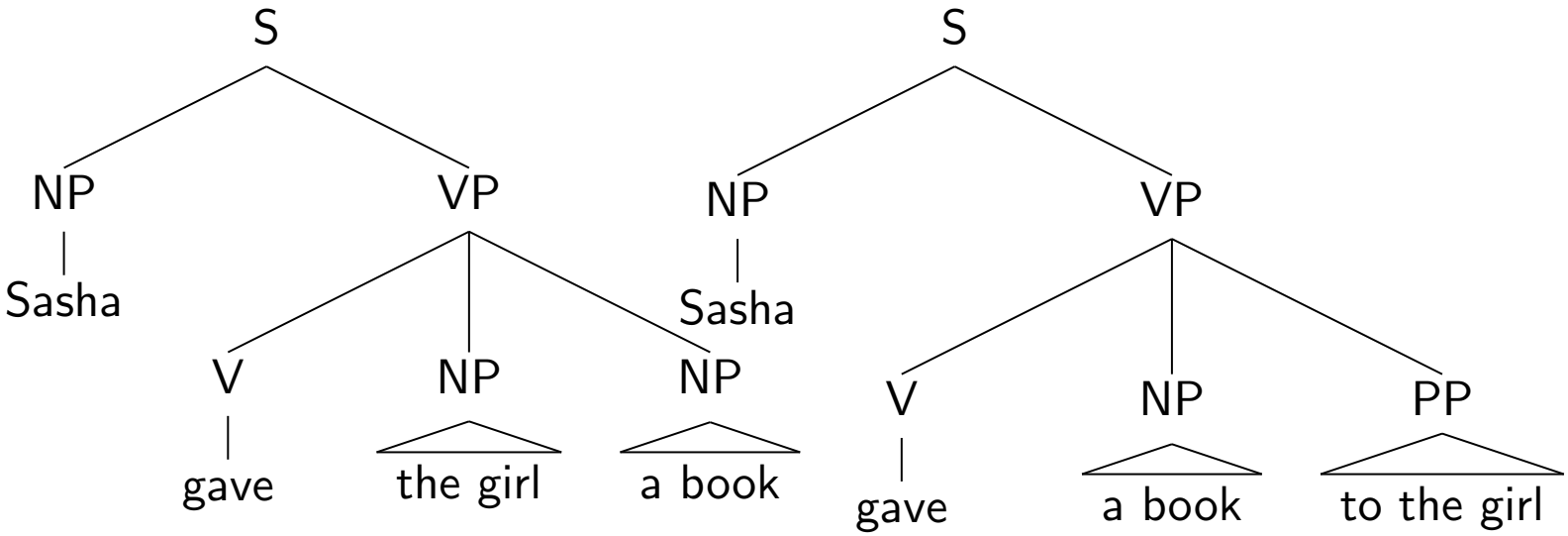
Consider these sentences. Two ways to say the same thing:



Shay Cohen, Accelerated Natural Language Processing (Slides)

Why dependencies??

Consider these sentences. Two ways to say the same thing:



- We only need a few phrase structure rules:
  $\texttt{S} \rightarrow \texttt{NP VP}$
  $\texttt{VP} \rightarrow \texttt{V NP NP}$
  $\texttt{VP} \rightarrow \texttt{V NP PP}$
  plus rules for NP and PP.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Equivalent sentences in Russian

- Russian uses morphology to mark relations between words:
  - knigu means book (kniga) as a direct object.
  - devochke means girl (devochka) as indirect object (to the girl).
- So we can have the same word orders as English:
  - Sasha dal devochke knigu
  - Sasha dal knigu devochke

Shay Cohen, Accelerated Natural Language Processing (Slides)

Equivalent sentences in Russian

- Russian uses morphology to mark relations between words:
  - knigu means book (kniga) as a direct object.
  - devochke means girl (devochka) as indirect object (to the girl).
- So we can have the same word orders as English:
  - Sasha dal devochke knigu
  - Sasha dal knigu devochke
- But also many others!
  - Sasha devochke dal knigu
  - Devochke dal Sasha knigu
  - Knigu dal Sasha devochke

Shay Cohen, Accelerated Natural Language Processing (Slides)

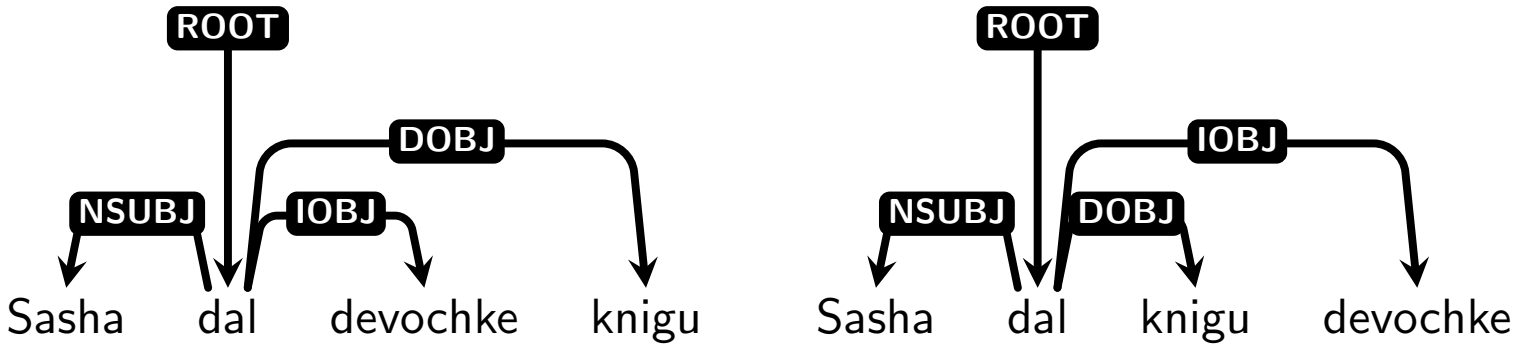
Phrase structure vs dependencies

- In languages with **free word order**, phrase structure (constituency) grammars don't make as much sense.
  - E.g., we would need both $\texttt{S} \rightarrow \texttt{NP VP}$ and $\texttt{S} \rightarrow \texttt{VP NP}$, etc. Not very informative about what's really going on.

Shay Cohen, Accelerated Natural Language Processing (Slides)

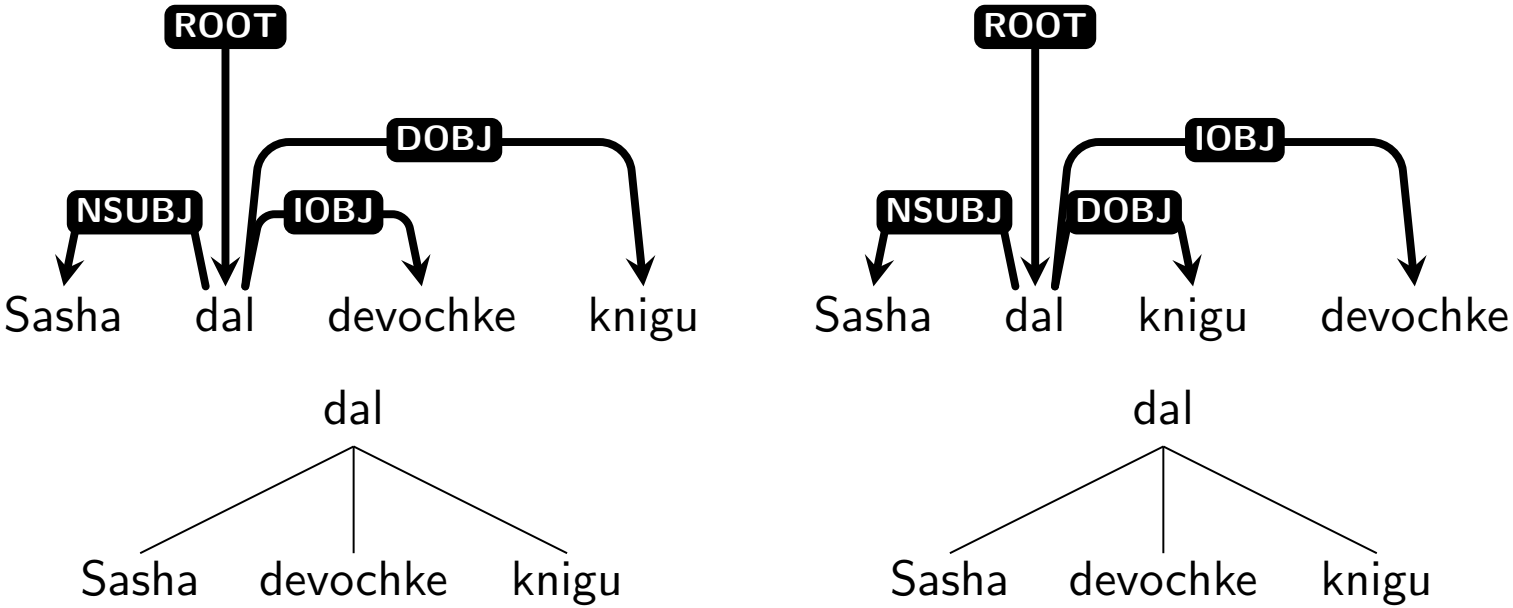
Phrase structure vs dependencies

- In languages with **free word order**, phrase structure (constituency) grammars don't make as much sense.
  - E.g., we would need both $\texttt{S} \rightarrow \texttt{NP VP}$ and $\texttt{S} \rightarrow \texttt{VP NP}$, etc. Not very informative about what's really going on.
- In contrast, the dependency relations stay constant:



Phrase structure vs dependencies

- Even more obvious if we just look at the trees without word order:



Shay Cohen, Accelerated Natural Language Processing (Slides)

Pros and cons

- Sensible framework for free word order languages.
- Identifies syntactic relations directly. (using CFG, how would you identify the subject of a sentence?)
- Dependency pairs/chains can make good features in classifiers, for information extraction, etc.
- Parsers can be very fast (coming up...)

But

- The assumption of asymmetric binary relations isn't always right... e.g., how to parse dogs and cats?

Shay Cohen, Accelerated Natural Language Processing (Slides)

How do we annotate dependencies?

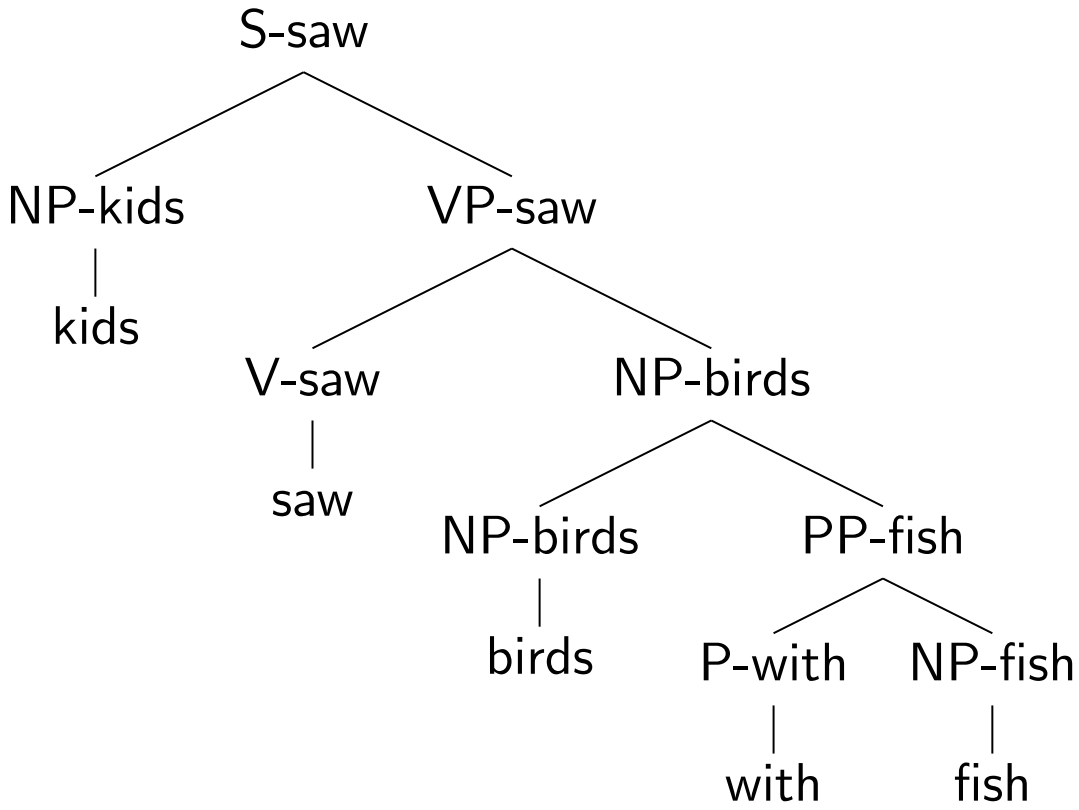
Two options:

1. Annotate dependencies directly.
2. Convert phrase structure annotations to dependencies. (Convenient if we already have a phrase structure treebank.)

Next slides show how to convert, assuming we have head-finding rules for our phrase structure trees.

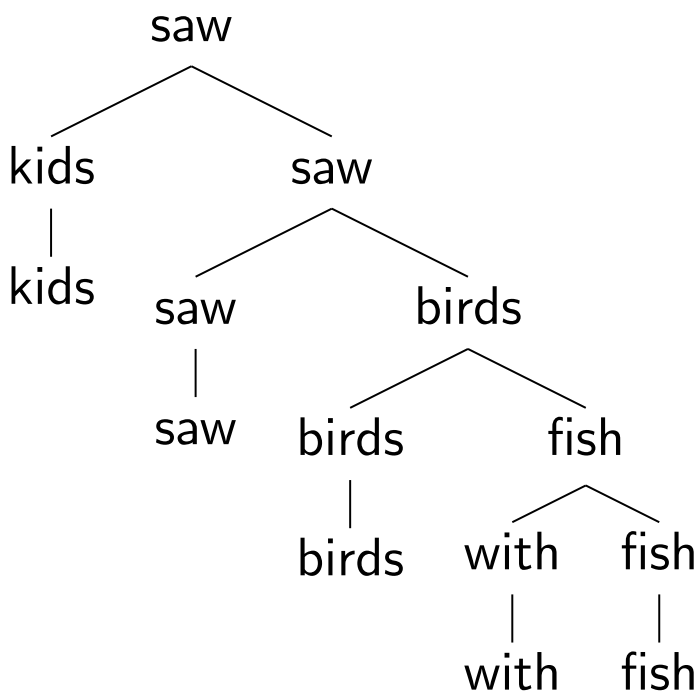
Shay Cohen, Accelerated Natural Language Processing (Slides)

Lexicalized Constituency Parse



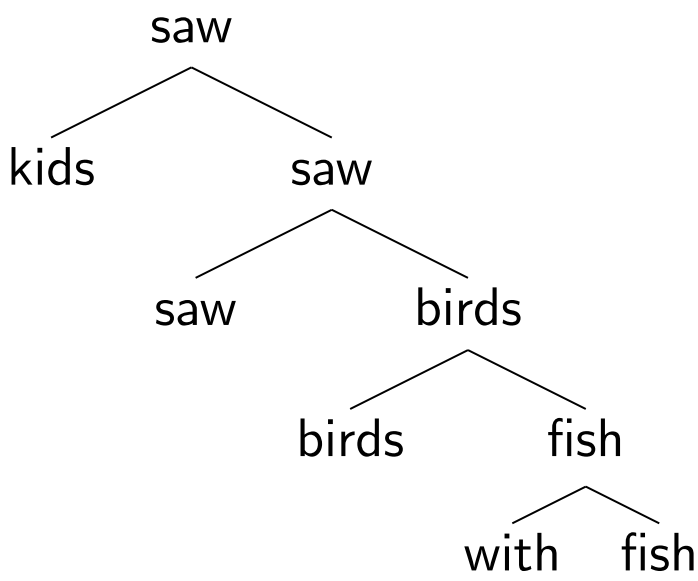
Shay Cohen, Accelerated Natural Language Processing (Slides)

. . . remove the phrasal categories. . .



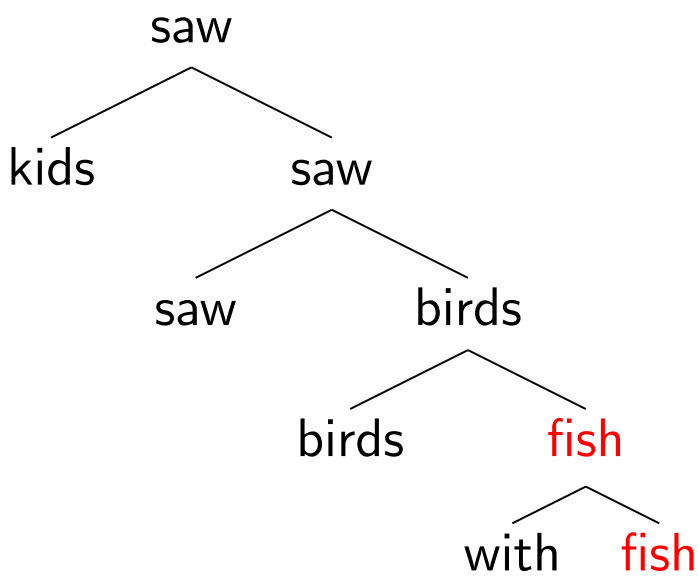
Shay Cohen, Accelerated Natural Language Processing (Slides)

. . . remove the (duplicated) terminals. . .



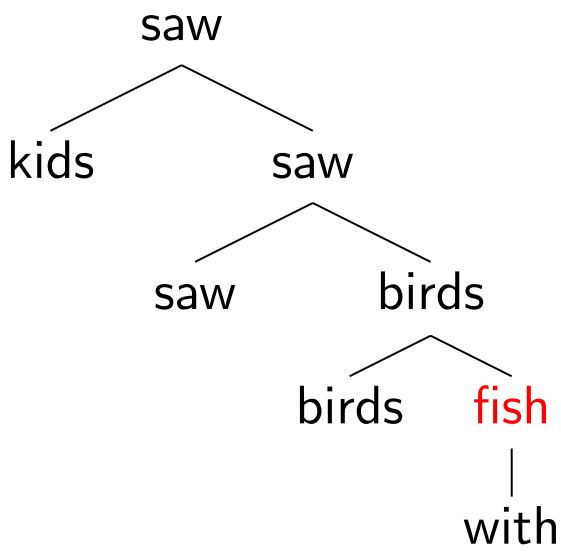
Shay Cohen, Accelerated Natural Language Processing (Slides)

. . . and collapse chains of duplicates. . .



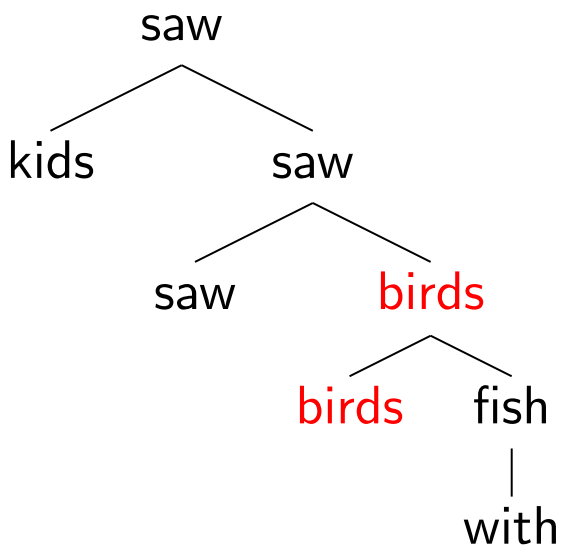
Shay Cohen, Accelerated Natural Language Processing (Slides)

. . . and collapse chains of duplicates. . .



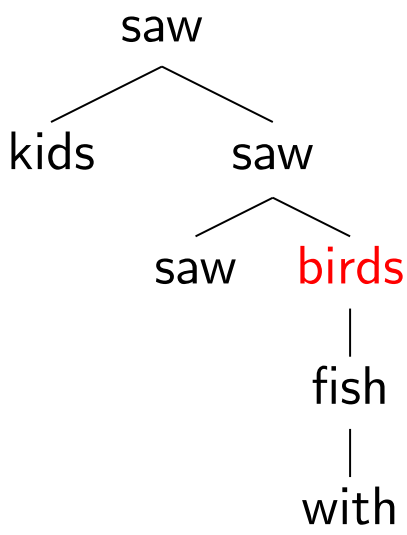
Shay Cohen, Accelerated Natural Language Processing (Slides)

. . . and collapse chains of duplicates. . .



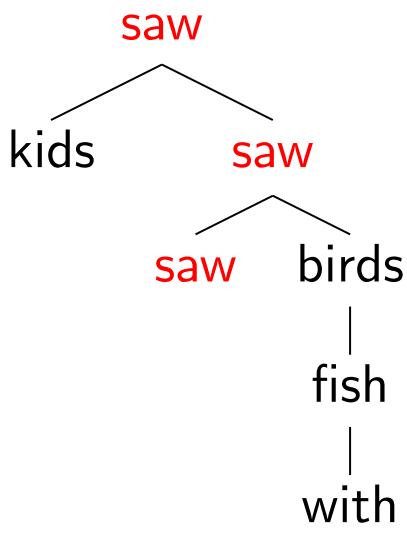
Shay Cohen, Accelerated Natural Language Processing (Slides)

. . . and collapse chains of duplicates. . .



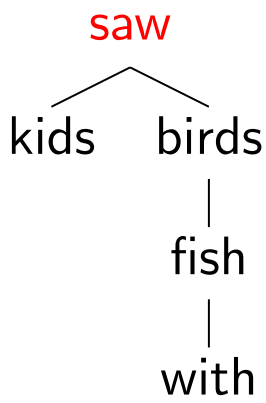
Shay Cohen, Accelerated Natural Language Processing (Slides)

. . . and collapse chains of duplicates. . .



Shay Cohen, Accelerated Natural Language Processing (Slides)

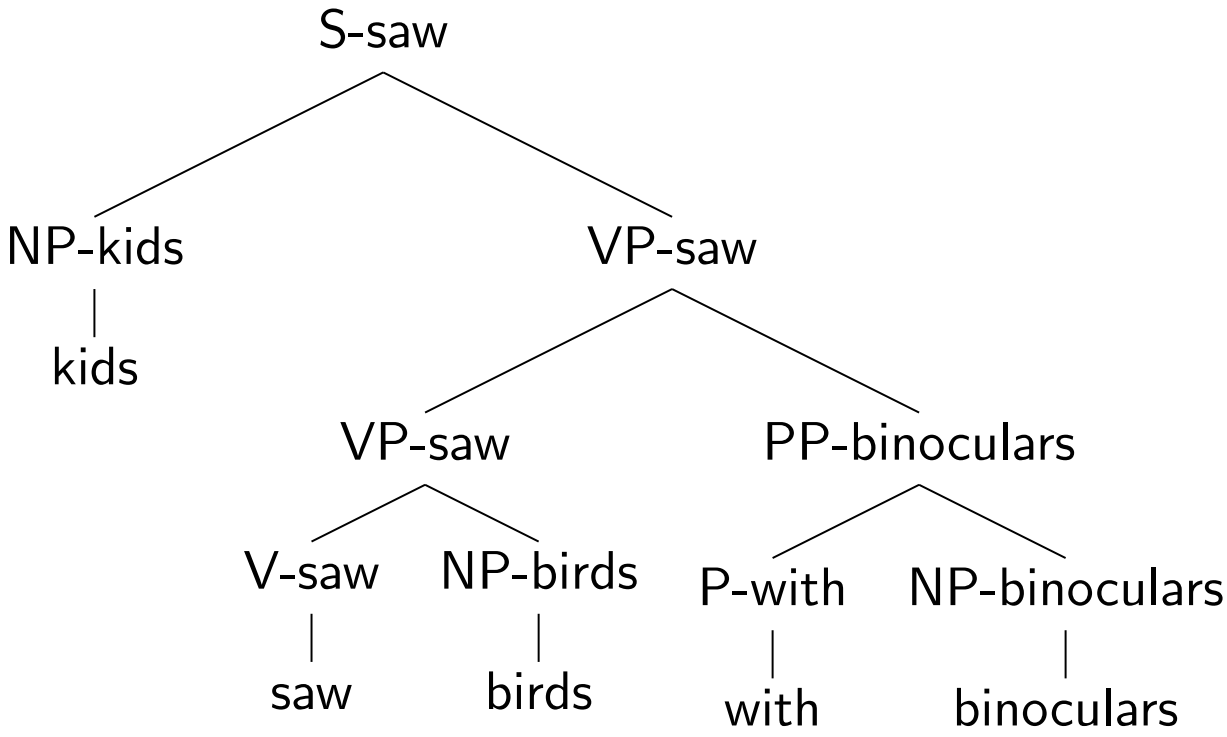
. . . done!



Shay Cohen, Accelerated Natural Language Processing (Slides)

Constituency Tree → Dependency Tree

We saw how the **lexical head** of the phrase can be used to collapse down to a dependency tree:

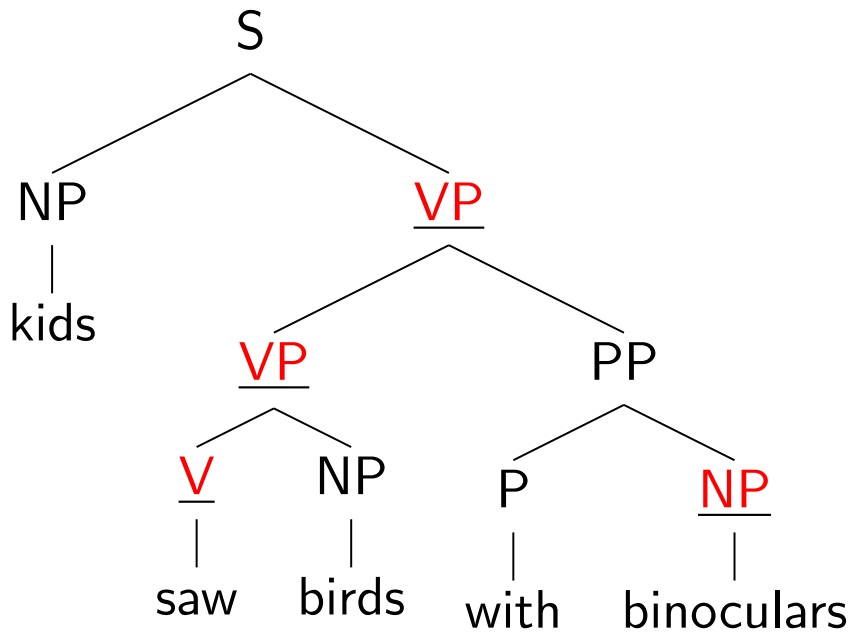


- But how can we find each phrase's head in the first place?

Shay Cohen, Accelerated Natural Language Processing (Slides)

Head Rules

The standard solution is to use **head rules**: for every non-unary (P)CFG production, designate one RHS nonterminal as containing the head. S → NP VP, VP → VP PP, PP → P NP (content head), etc.

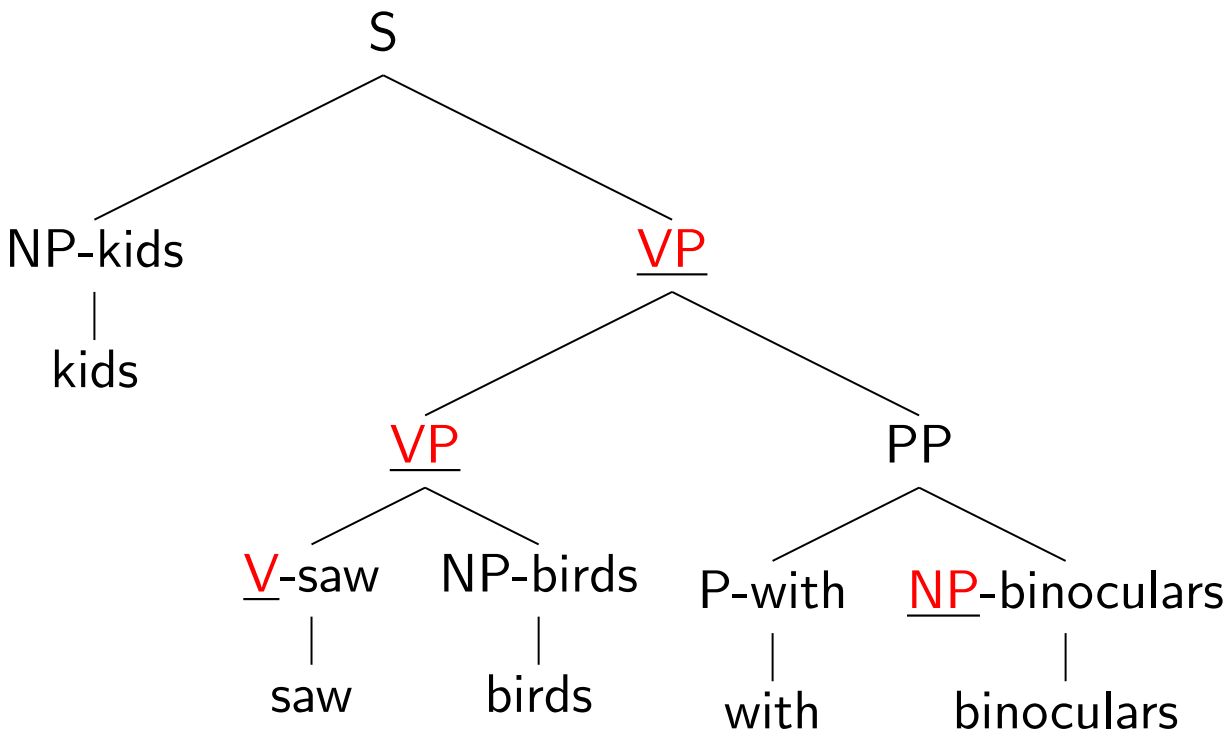


- Heuristics to scale this to large grammars: e.g., within an NP, last immediate N child is the head.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Head Rules

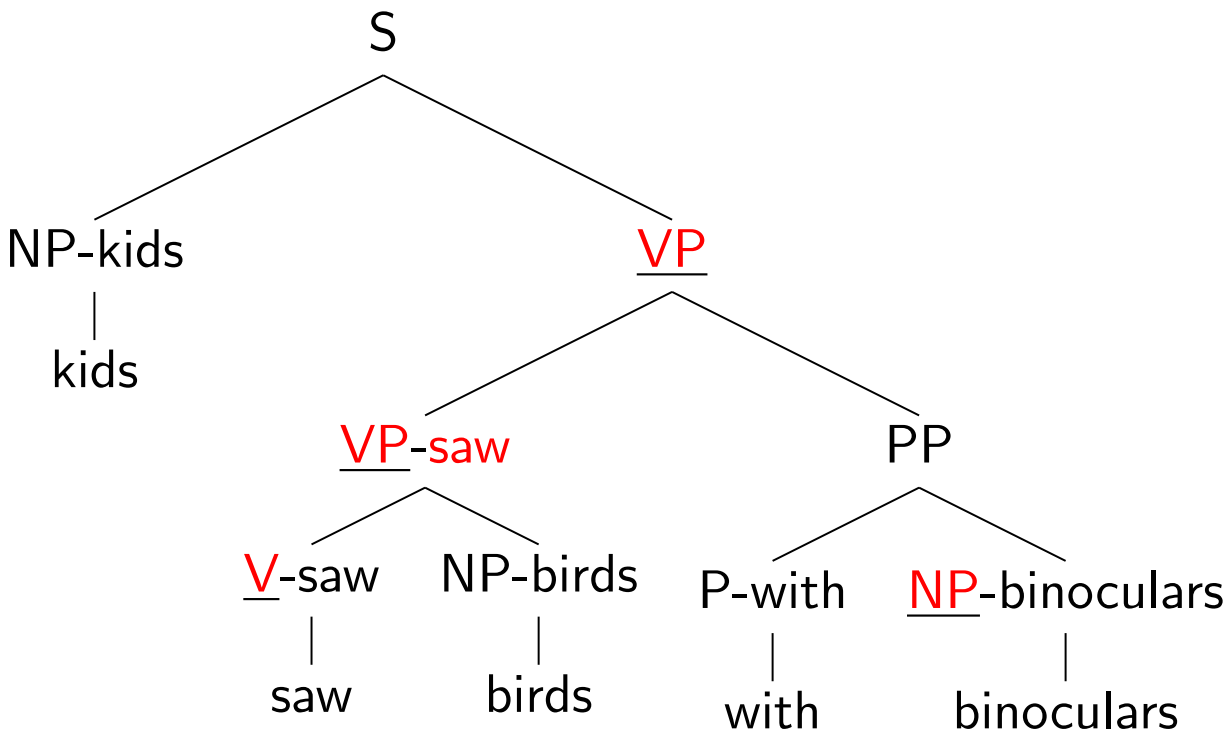
Then, propagate heads up the tree:



Shay Cohen, Accelerated Natural Language Processing (Slides)

Head Rules

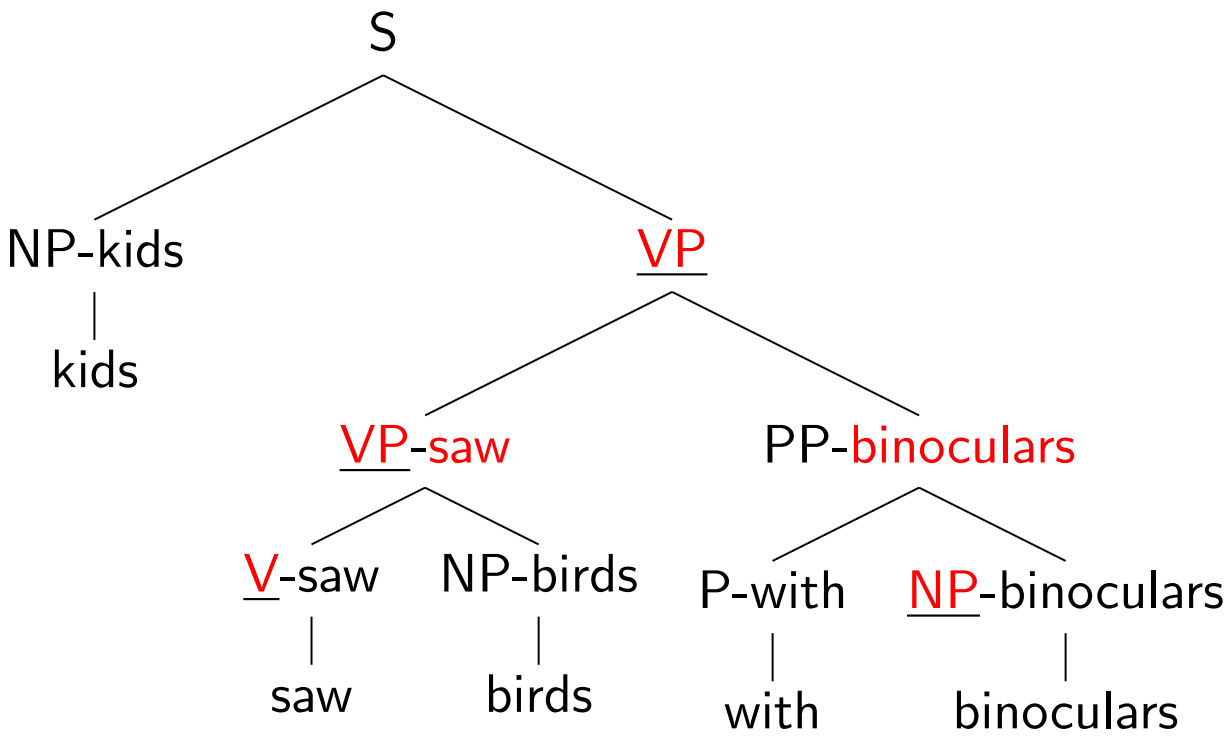
Then, propagate heads up the tree:



Shay Cohen, Accelerated Natural Language Processing (Slides)

Head Rules

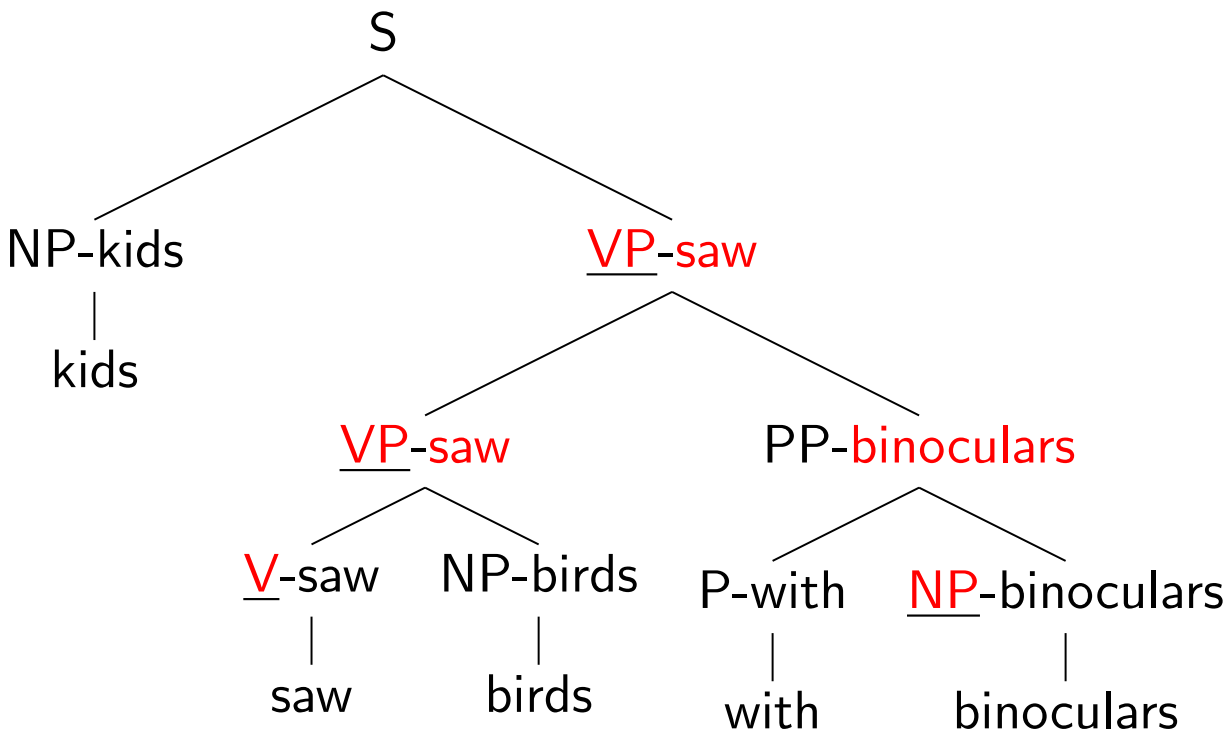
Then, propagate heads up the tree:



Shay Cohen, Accelerated Natural Language Processing (Slides)

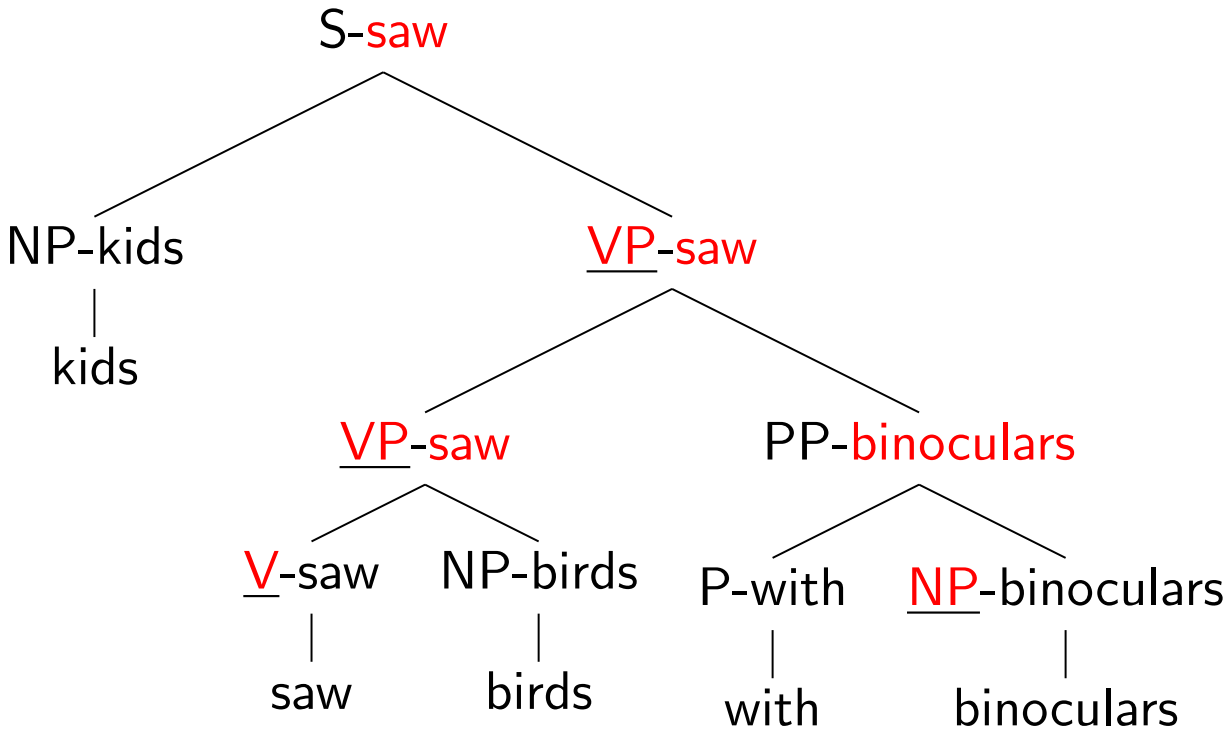
Head Rules

Then, propagate heads up the tree:



Head Rules

Then, propagate heads up the tree:

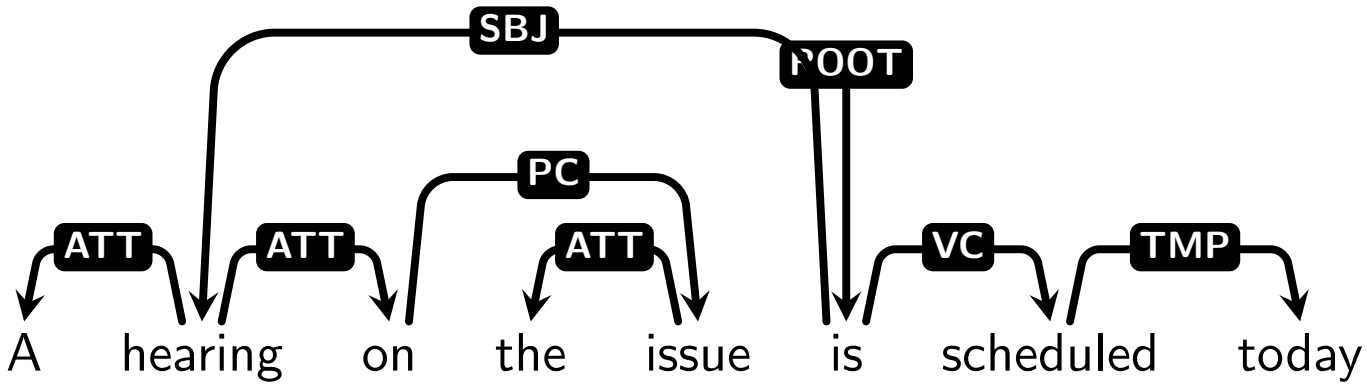


Shay Cohen, Accelerated Natural Language Processing (Slides)

Projectivity

If we convert constituency parses to dependencies, all the resulting trees will be **projective**.

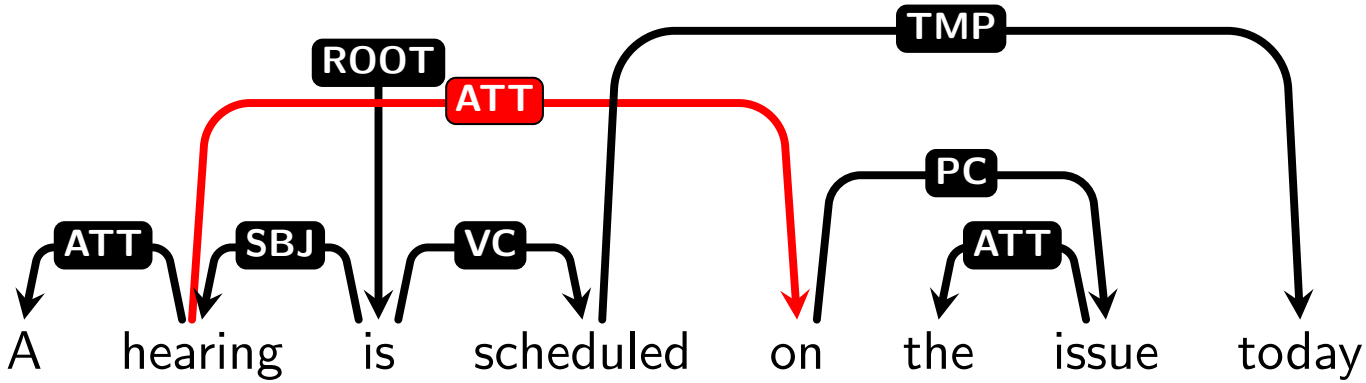
- Every subtree (node and all its descendants) occupies a *contiguous span* of the sentence.
- = the parse can be drawn over the sentence w/ no crossing edges.



Shay Cohen, Accelerated Natural Language Processing (Slides)

Nonprojectivity

But some sentences are **nonprojective**.



- We'll only get these annotations right if we directly annotate the sentences (or correct the converted parses).
- Nonprojectivity is rare in English, but common in many languages.
- Nonprojectivity presents problems for parsing algorithms.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Dependency Parsing

Some of the algorithms you have seen for PCFGs can be adapted to dependency parsing.

- **CKY** can be adapted, though efficiency is a concern: obvious approach is $O(Gn^5)$; Eisner algorithm brings it down to $O(Gn^3)$
  - N. Smith's slides explaining the Eisner algorithm: `http://courses.cs.washington.edu/courses/cse517/16wi/slides/an-dep-slides.pdf`
- **Shift-reduce**: more efficient, doesn't even require a grammar!

Shay Cohen, Accelerated Natural Language Processing (Slides)

Recall: shift-reduce parser with CFG

- Same example grammar and sentence.
- Operations:
  - Reduce (R)
  - Shift (S)
  - Backtrack to step n (Bn)
- Note that at 9 and 11 we skipped over backtracking to 7 and 5 respectively as there were actually no choices to be made at those points.

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	R	DT VP	bit
6	S	DT VP bit	
7	R	DT VP V	
8	R	DT VP VP	
9	B6	DT VP bit	
10	R	DT VP NN	
11	B4	DT V	bit
12	S	DT V bit	
13	R	DT V V	
14	R	DT V VP	
15	B3	DT dog	bit
16	R	DT NN	bit
17	R	NP	bit
...			

Shay Cohen, Accelerated Natural Language Processing (Slides)

Transition-based Dependency Parsing

The **arc-standard** approach parses input sentence $w_1 \ldots w_N$ using two types of **reduce** actions (three actions altogether):

- **Shift:** Read next word w_i from input and push onto the stack.
- **LeftArc:** Assign head-dependent relation $s_2 \leftarrow s_1$; pop s_2
- **RightArc:** Assign head-dependent relation $s_2 \rightarrow s_1$; pop s_1

where s_1 and s_2 are the top and second item on the stack, respectively. (So, s_2 *preceded* s_1 in the input sentence.)

Shay Cohen, Accelerated Natural Language Processing (Slides)

Example

Parsing Kim saw Sandy:

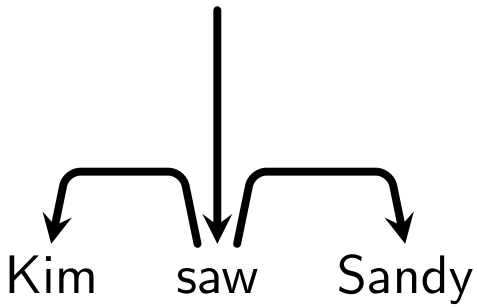
Step	←bot. Stacktop→	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]	Shift	
1	[root,Kim]	[saw,Sandy]	Shift	
2	[root,Kim,saw]	[Sandy]	LeftArc	Kim←saw
3	[root,saw]	[Sandy]	Shift	
4	[root,saw,Sandy]	[]	RightArc	saw→Sandy
5	[root,saw]	[]	RightArc	root→saw
6	[root]	[]	(done)	

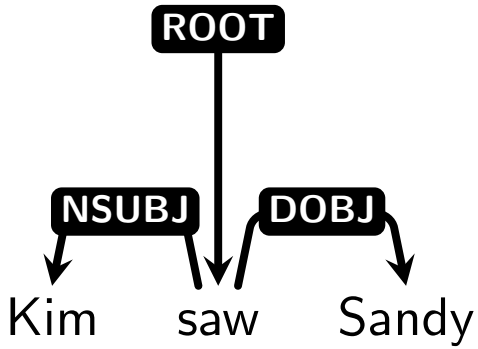
- Here, top two words on stack are also always adjacent in sentence. Not true in general! (See longer example in JM3.)

Shay Cohen, Accelerated Natural Language Processing (Slides)

Labelled dependency parsing

- These parsing actions produce **unlabelled** dependencies (left).
- For **labelled** dependencies (right), just use more actions: LeftArc(NSUBJ), RightArc(NSUBJ), LeftArc(DOBJ), ...





Shay Cohen, Accelerated Natural Language Processing (Slides)

Differences to constituency parsing

- Shift-reduce parser for CFG: not all sequences of actions lead to valid parses. Choose incorrect action $\rightarrow$ may need to backtrack.
- Here, all valid action sequences lead to valid parses.
  - Invalid actions: can't apply LeftArc with root as dependent; can't apply RightArc with root as head unless input is empty.
  - Other actions may lead to **incorrect** parses, but still **valid**.
- So, parser doesn't backtrack. Instead, tries to greedily predict the correct action at each step.
  - Therefore, dependency parsers can be very fast (linear time).
  - But need a good way to predict correct actions (next lecture).

Shay Cohen, Accelerated Natural Language Processing (Slides)

Notions of validity

- In constituency parsing, valid parse = grammatical parse.
  - That is, we first define a grammar, then use it for parsing.
- In dependency parsing, we don't normally define a grammar.
  - Valid parses are those with the properties on slide 4.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Summary: Transition-based Parsing

- **arc-standard** approach is based on simple shift-reduce idea.
- Can do labelled or unlabelled parsing, but need to train a **classifier** to predict next action, as we'll see next time.
- Greedy algorithm means time complexity is linear in sentence length.
- Only finds **projective** trees (without special extensions)
- Pioneering system: Nivre's MaltParser.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Alternative: Graph-based Parsing

- Global algorithm: From the fully connected directed graph of all possible edges, choose the best ones that form a tree.
- **Edge-factored** models: Classifier assigns a nonnegative score to each possible edge; **maximum spanning tree** algorithm finds the spanning tree with highest total score in $O(n^2)$ time.
- Pioneering work: McDonald's MSTParser
- Can be formulated as constraint-satisfaction with **integer linear programming** (Martins's TurboParser)
- Details in JM3, Ch 16.5 (optional).

Shay Cohen, Accelerated Natural Language Processing (Slides)

Graph-based vs. Transition-based vs. Conversion-based

- TB: Features in scoring function can look at any part of the stack; no optimality guarantees for search; linear-time; (classically) projective only
- GB: Features in scoring function limited by factorization; optimal search within that model; quadratic-time; no projectivity constraint
- CB: In terms of accuracy, sometimes best to first constituency-parse, then convert to dependencies (e.g., Stanford Parser). Slower than direct methods.

Shay Cohen, Accelerated Natural Language Processing (Slides)

Choosing a Parser: Criteria

- Target representation: constituency or dependency?
- Efficiency? In practice, both runtime and memory use.
- Incrementality: parse the whole sentence at once, or obtain partial left-to-right analyses/expectations?
- Retrainable system?
- Accuracy?

Shay Cohen, Accelerated Natural Language Processing (Slides)

Summary

- Constituency syntax: hierarchically nested phrases with categories like `NP`.
- Dependency syntax: trees whose edges connect words in the sentence. Edges often labeled with relations like `nsubj`.
- Can convert constituency to dependency parse using head rules.
- For projective trees, transition-based parsing is very fast and can be very accurate.
- Google "online dependency parser".
  Try out the Stanford parser and SEMAFOR!

Shay Cohen, Accelerated Natural Language Processing (Slides)

Content

1 Grammars and syntax parsing

2 Parsing with context free grammars

3 Parsing with probabilistic context free grammars

4 Dependency parsing

5 Parsing with neural networks

Parsing with neural networks

- Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks, EMNLP2014

  (transition-based dependency parser)
- Timothy Dozat, Peng Qi, and Chris Manning. Stanford's Graph-based Neural Dependency Parser at the CoNLL 2017 Shared Task. In CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies, 2017.

  (graph-based dependency parser)

Parsing with neural networks

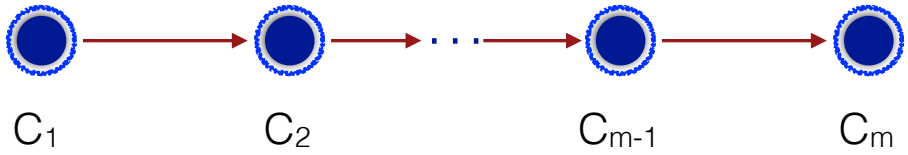
- Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks, EMNLP2014

  (transition-based dependency parser)
- Timothy Dozat, Peng Qi, and Chris Manning. Stanford's Graph-based Neural Dependency Parser at the CoNLL 2017 Shared Task. In CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies, 2017.

  (graph-based dependency parser)

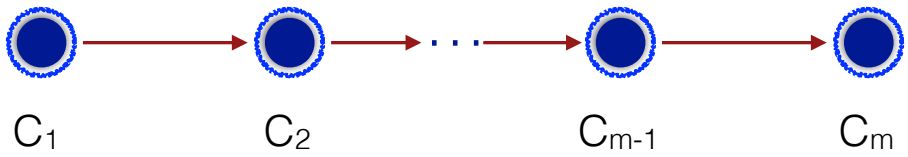
(Nivre et al, 2004)

Greedy Transition-based Parsing

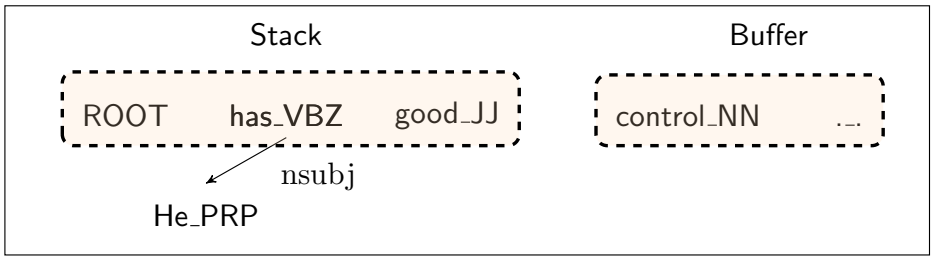


Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Greedy Transition-based Parsing

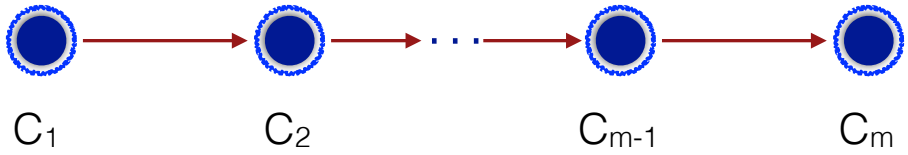


- A configuration = a stack, a buffer and some dependency arcs

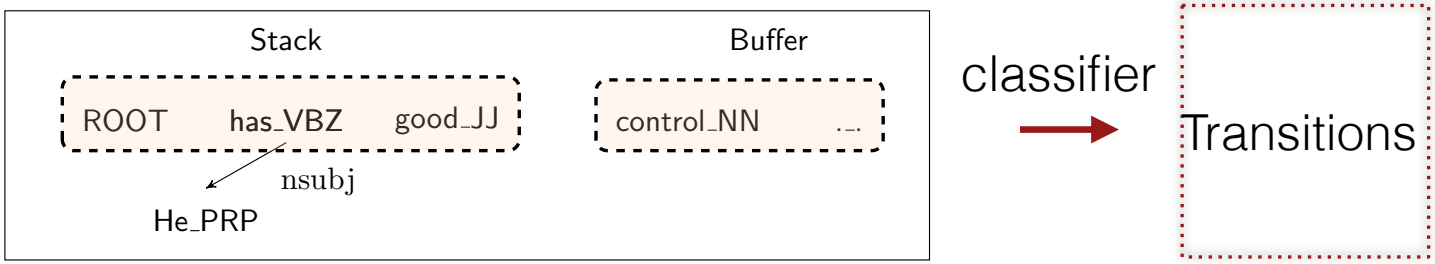


Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Greedy Transition-based Parsing

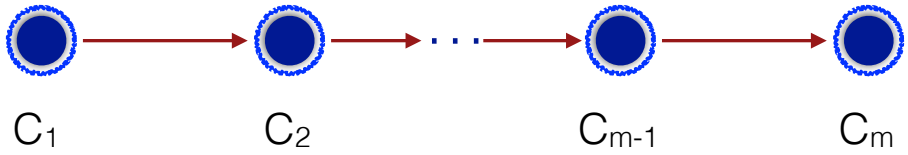


- A configuration = a stack, a buffer and some dependency arcs

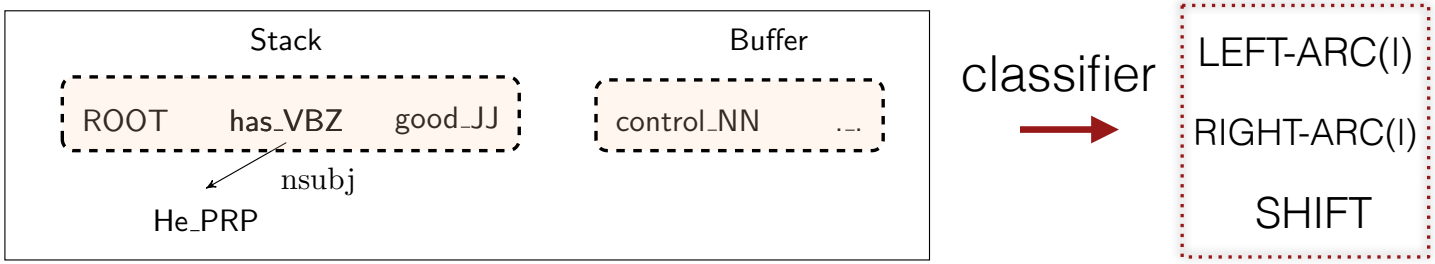


Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Greedy Transition-based Parsing



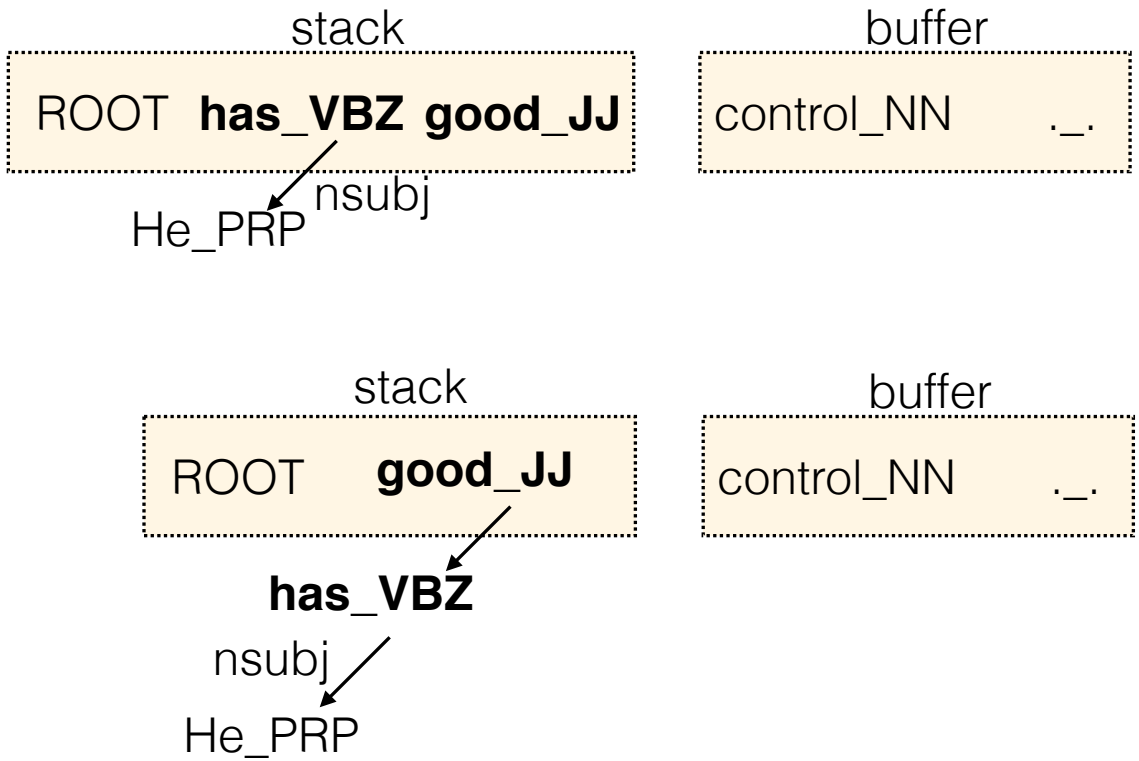
- A configuration = a stack, a buffer and some dependency arcs



- We employ the arc-standard system.

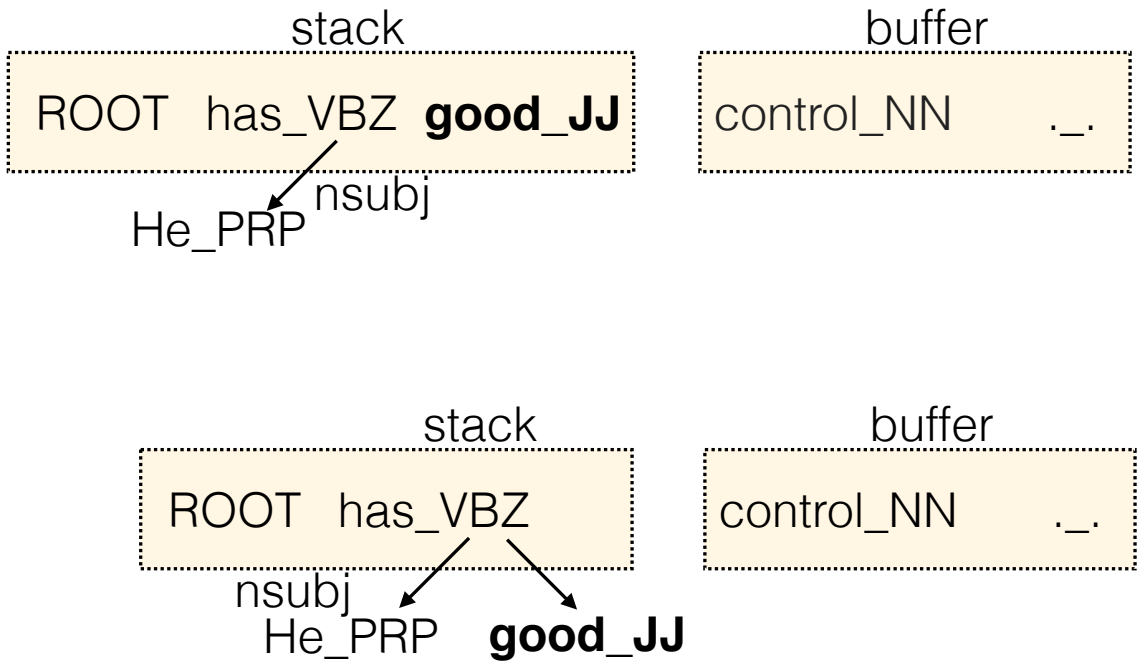
Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

LEFT-ARC (I)



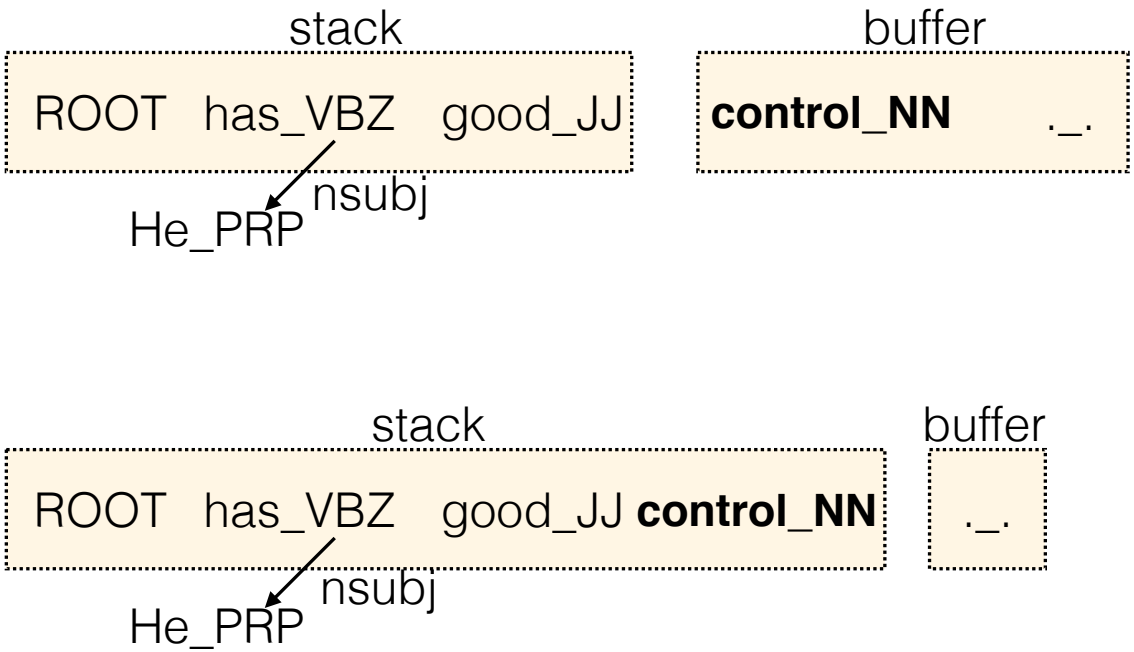
Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

RIGHT-ARC (l)



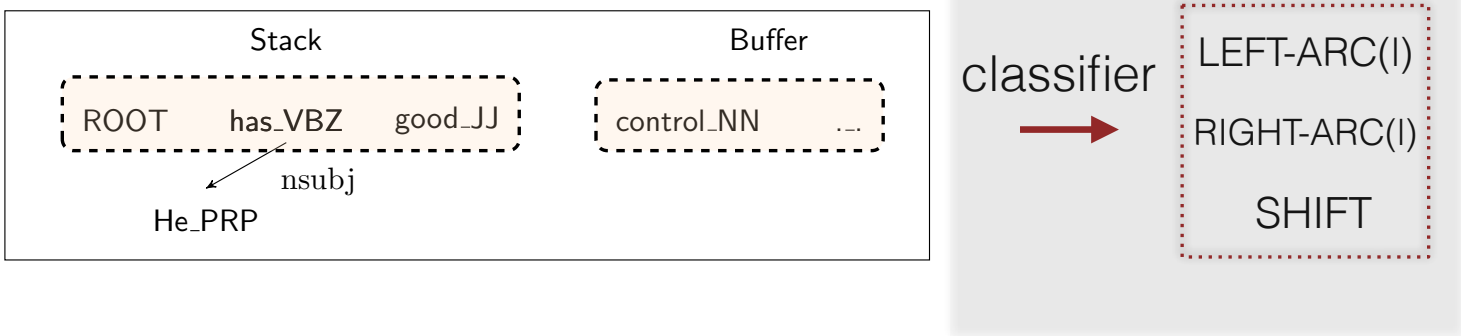
Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

SHIFT



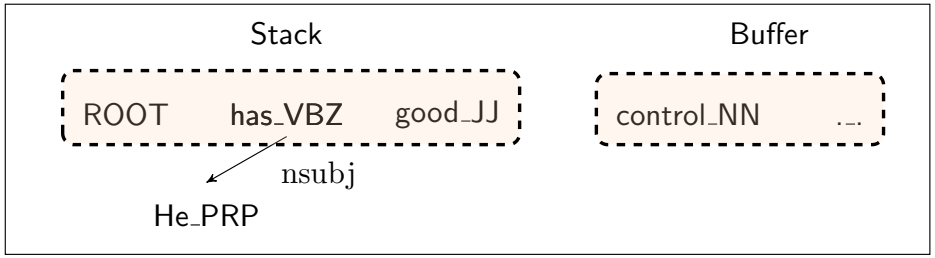
Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Greedy Transition-based Parsing



Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Traditional Features



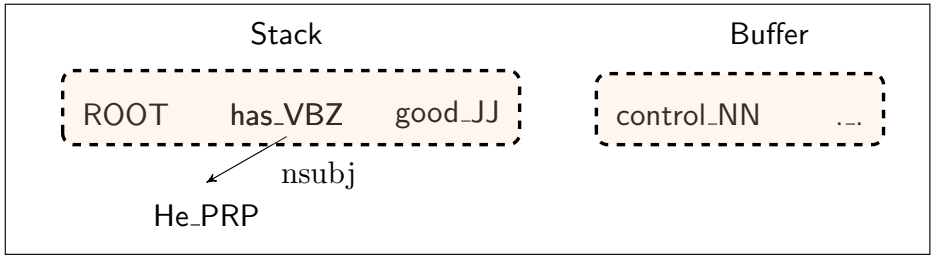
binary, sparse
dim = 10^6 ~ 10^7

0	0	0	1	0	0	1	0	...	0	0	1	0

Feature templates: usually a combination of **1 ~ 3** elements from the configuration.

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Traditional Features



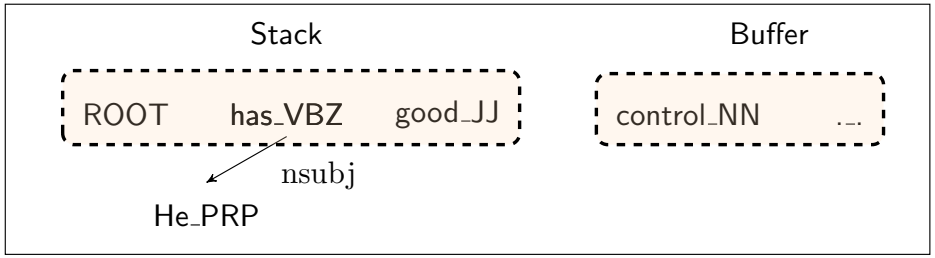
binary, sparse
dim =10^6 ~ 10^7

Indicator features

$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Traditional Features



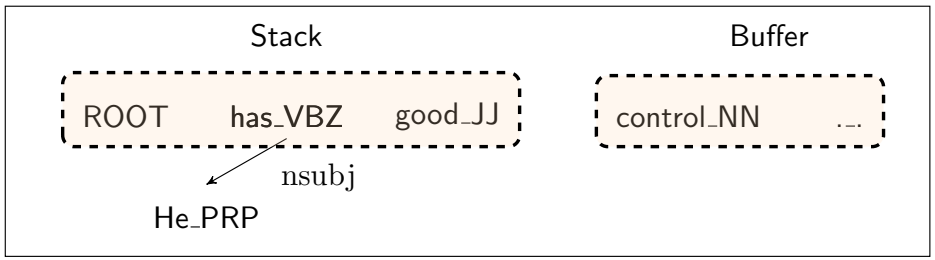
binary, sparse
dim =10^6 ~ 10^7

Indicator features

$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Traditional Features



binary, sparse
dim =10^6 ~ 10^7

Indicator features

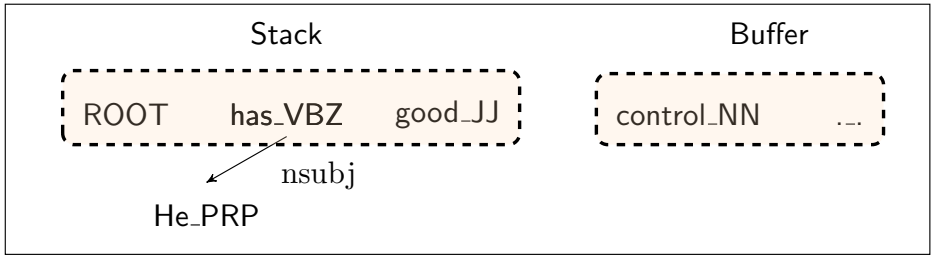
word part-of-speech tag

$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

dep. label

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Traditional Features



binary, sparse
dim =10^6 ~ 10^7

Indicator features

$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

leftmost child

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator Features Revisited

$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator Features Revisited

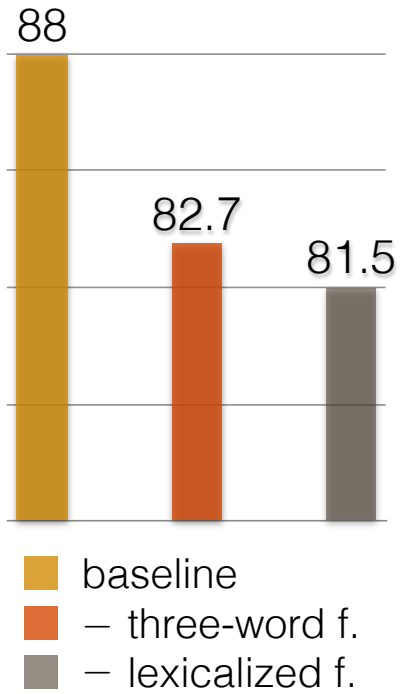
- Problem #1: sparse
  - lexicalized features
  - high-order interaction features

$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator Features Revisited

- Problem #1: sparse
  - lexicalized features
  - high-order interaction features



$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator Features Revisited

- Problem #1: sparse

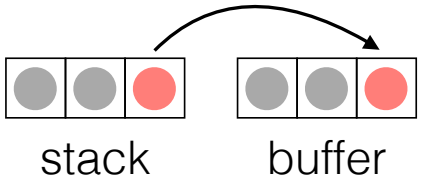
$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator Features Revisited

- Problem #1: sparse
- Problem #2: incomplete

Unavoidable in hand-crafted feature templates.



~~RIGHT-ARC~~

$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator Features Revisited

- Problem #1: sparse
- Problem #2: incomplete

$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator Features Revisited

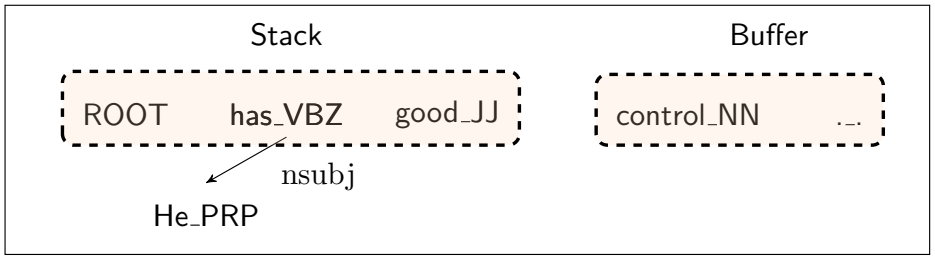
- Problem #1: sparse
- Problem #2: incomplete
- Problem #3: computationally expensive

More than 95% of parsing time is consumed by feature computation.

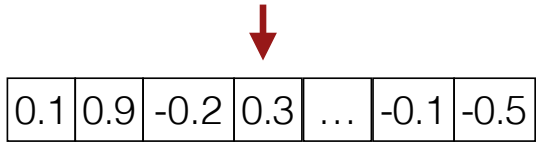
$$s_2.w = \text{has} \wedge s_2.t = \text{VBZ}$$
$$s_1.w = \text{good} \wedge s_1.t = \text{JJ} \wedge b_1.w = \text{control}$$
$$lc(s_2).t = \text{PRP} \wedge s_2.t = \text{VBZ} \wedge s_1.t = \text{JJ}$$
$$lc(s_2).w = \text{He} \wedge lc(s_2).l = \text{nsubj} \wedge s_2.w = \text{has}$$

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator Features Revisited



dense
dim = 200

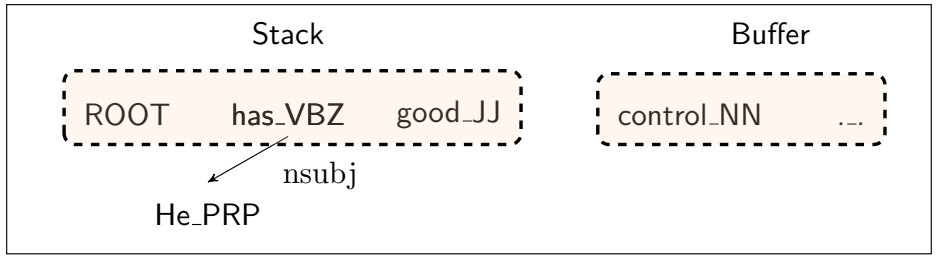


Our Solution: Neural Networks!

Learn a **dense** and **compact** feature representation

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

The Challenge



dense
dim = 200

- How to encode all the available information?
- How to model high-order features?

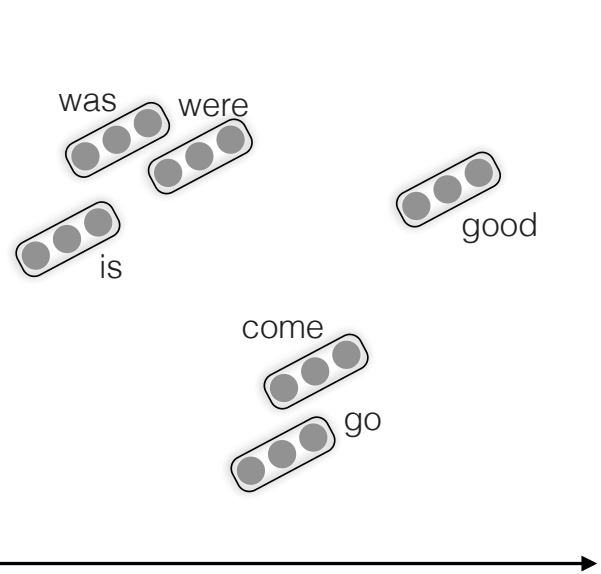
Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Distributed Representations

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Distributed Representations

- We represent each word as a d-dimensional dense vector (i.e., word embeddings).
  - Similar words expect to have close vectors.



Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Distributed Representations

- We represent each word as a d-dimensional dense vector (i.e., word embeddings).
  - Similar words expect to have close vectors.
- Meanwhile, part-of-speech tags and dependency labels are also represented as d-dimensional vectors.
  - POS and dependency embeddings.
  - The smaller discrete sets also exhibit many semantical similarities.

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Distributed Representations

- We represent each word as a d-dimensional dense vector (i.e., word embeddings).
  - Similar words expect to have close vectors.
- Meanwhile, part-of-speech tags and dependency labels are also represented as d-dimensional vectors.
  - POS and dependency embeddings.
  - The smaller discrete sets also exhibit many semantical similarities.

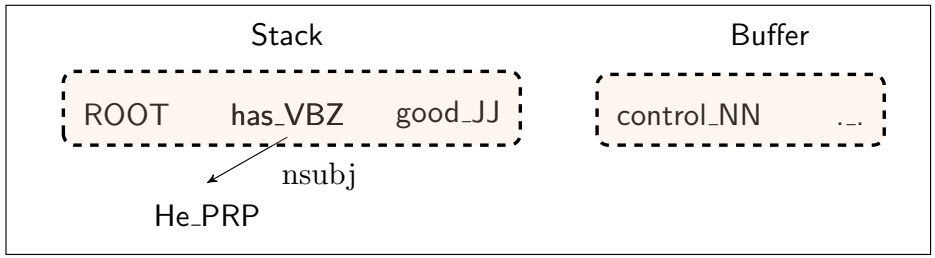
NNS (plural noun) should be close to NN (singular noun).

num (numerical modifier) should be close to amod (adjective modifier).

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Extracting Tokens from Configuration

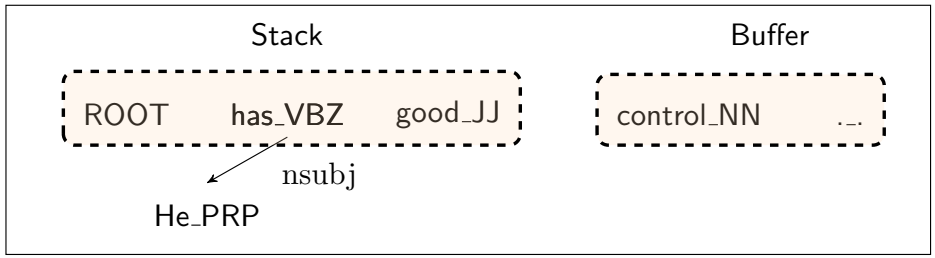
- We extract a set of tokens based on the positions:



Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Extracting Tokens from Configuration

- We extract a set of tokens based on the positions:

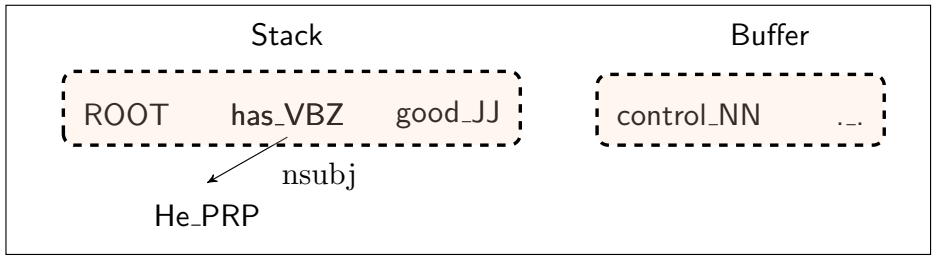


s_1
s_2
b_1
$lc(s_1)$
$rc(s_1)$
$lc(s_2)$
$rc(s_2)$
...

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Extracting Tokens from Configuration

- We extract a set of tokens based on the positions:



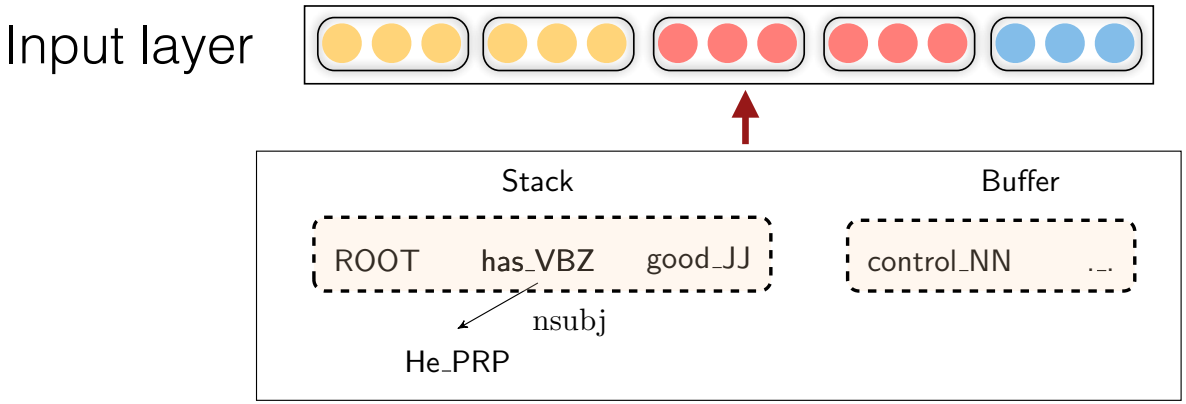
	word	POS	dep.
s_1	good	JJ	
s_2	has	VBZ	
b_1	control	NN	
$lc(s_1)$	∅	∅	∅
$rc(s_1)$	∅	∅	∅
$lc(s_2)$	He	PRP	nsubj
$rc(s_2)$	∅	∅	∅
…	…	…	…

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Model Architecture

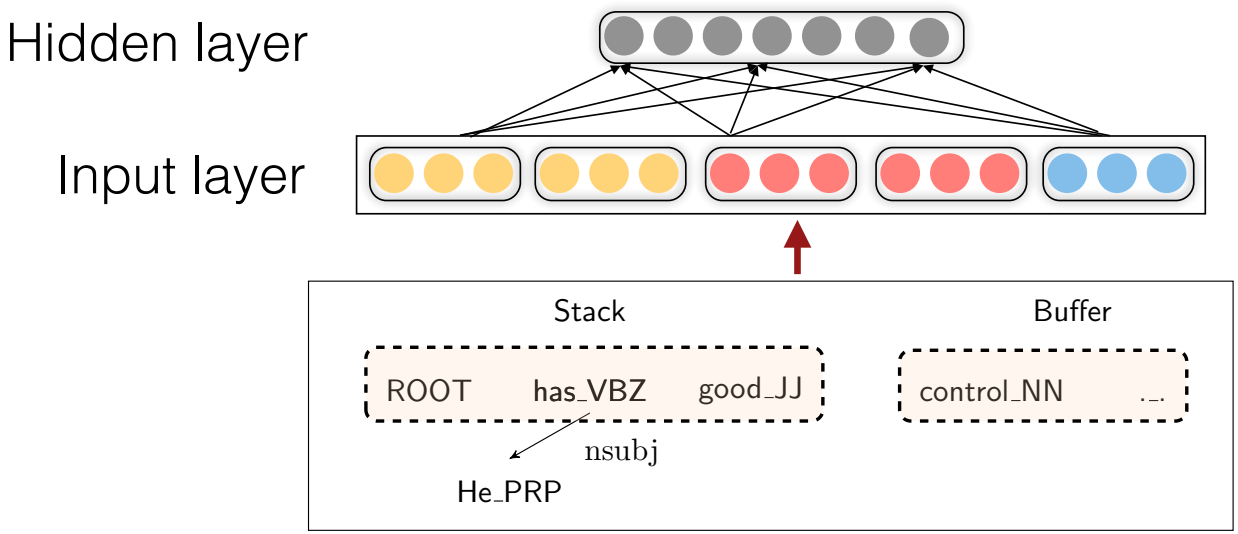
Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Model Architecture



Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

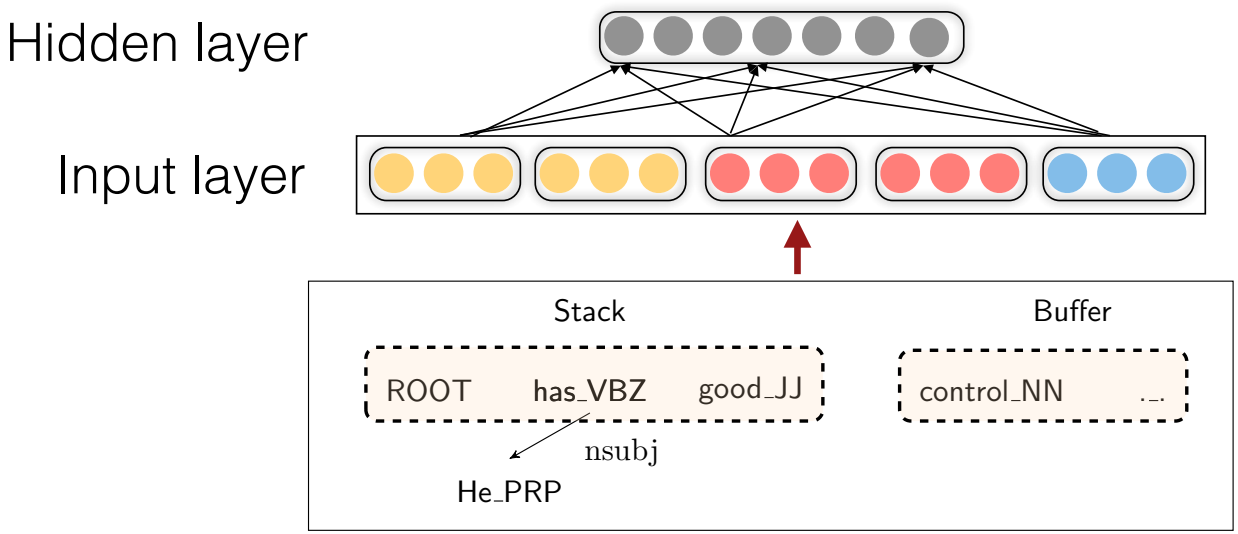
Model Architecture



Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

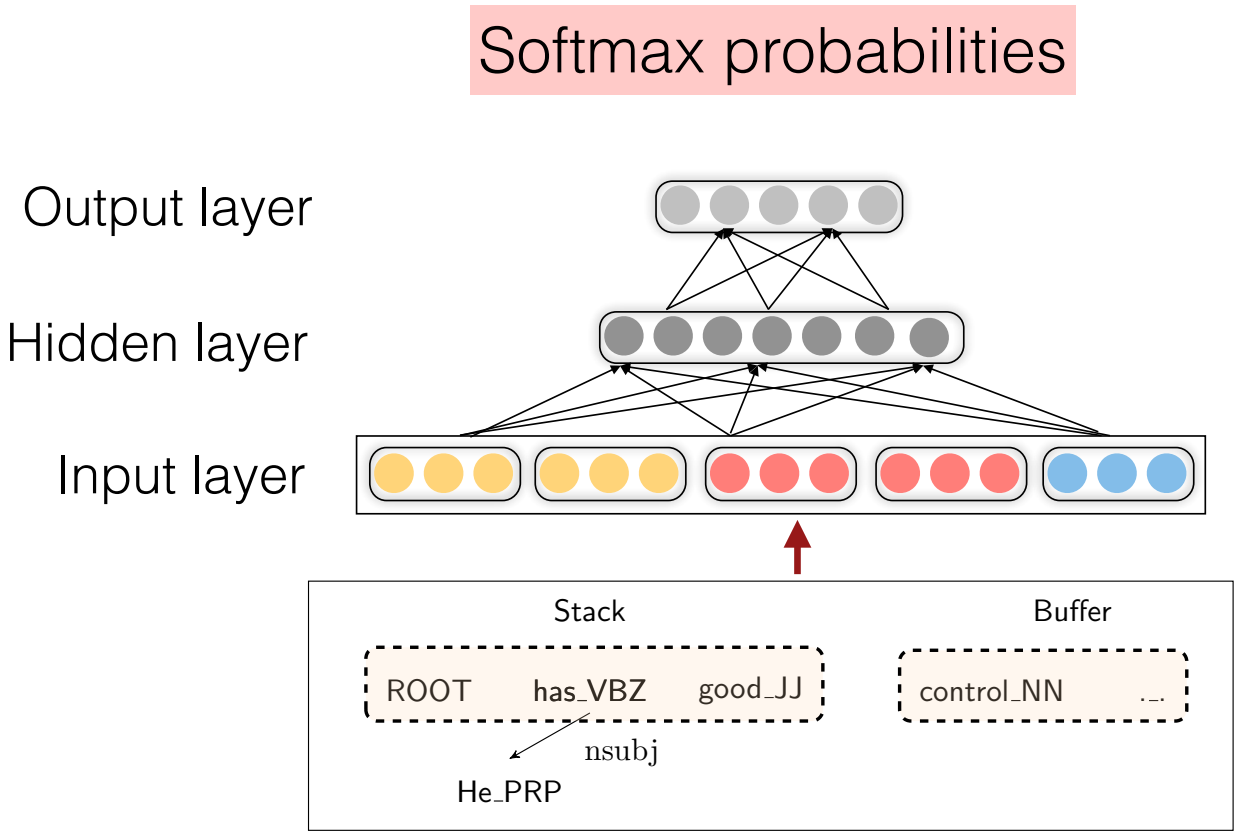
Model Architecture

Cube activation function: $g(x) = x^3$



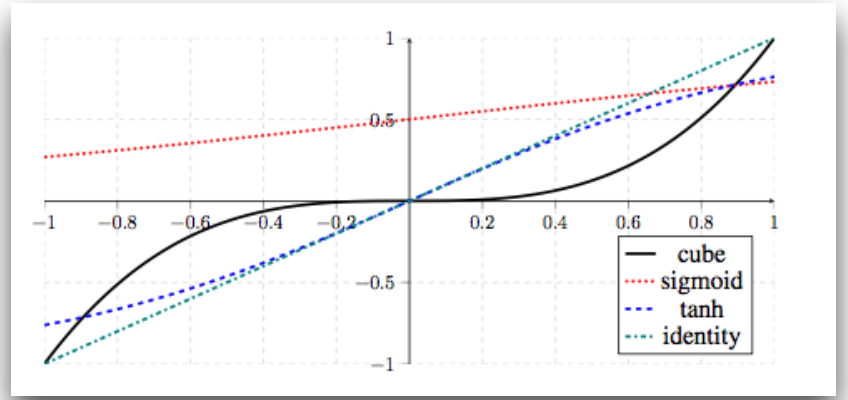
Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Model Architecture



Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Cube Activation Function



$$g(w_1x_1 + \ldots + w_mx_m + b) =$$
$$\sum_{i,j,k}(w_iw_jw_k)x_ix_jx_k + \sum_{i,j} b(w_iw_j)x_ix_j \ldots$$

Better capture the **interaction** terms!

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Training

- Generating training examples using a fixed oracle.
- **Training objective**: cross entropy loss
- Back-propagation to all embeddings.
- **Initialization**:
  - Word embeddings from pre-trained word vectors.
  - Random initialization for others.

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Indicator vs. Dense Features

- Problem #1: sparse

  Distributed representations can capture similarities.

- Problem #2: incomplete

  We don't need to enumerate the combinations.
  Cube non-linearity can learn combinations automatically.

- Problem #3: computationally expensive

  String concatenation + look-up in a big table $\Longrightarrow$ matrix operations. Pre-computation trick can speed up.

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Experimental Setup

- **Datasets**
  - English Penn Treebank (PTB)
  - Chinese Penn Treebank (CTB)
- **Representations**
  - CoNLL representations (CD) for PTB and CTB
  - Stanford Dependencies V3.3.0 (SD) for PTB
- **Part-of-speech tags**:
  - Stanford POS tagger for PTB (97.3% accuracy)
  - Gold tags for CTB

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Details

- Embedding size = 50
- Hidden size = 200
- Use mini-batched AdaGrad for optimization ($\alpha = 0.01$)
- Use 0.5 dropout on hidden layer.

- Pre-trained word vectors:
  - C & W for English
  - Word2vec for Chinese

- We use a rich set of 18 tokens from the configuration.

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

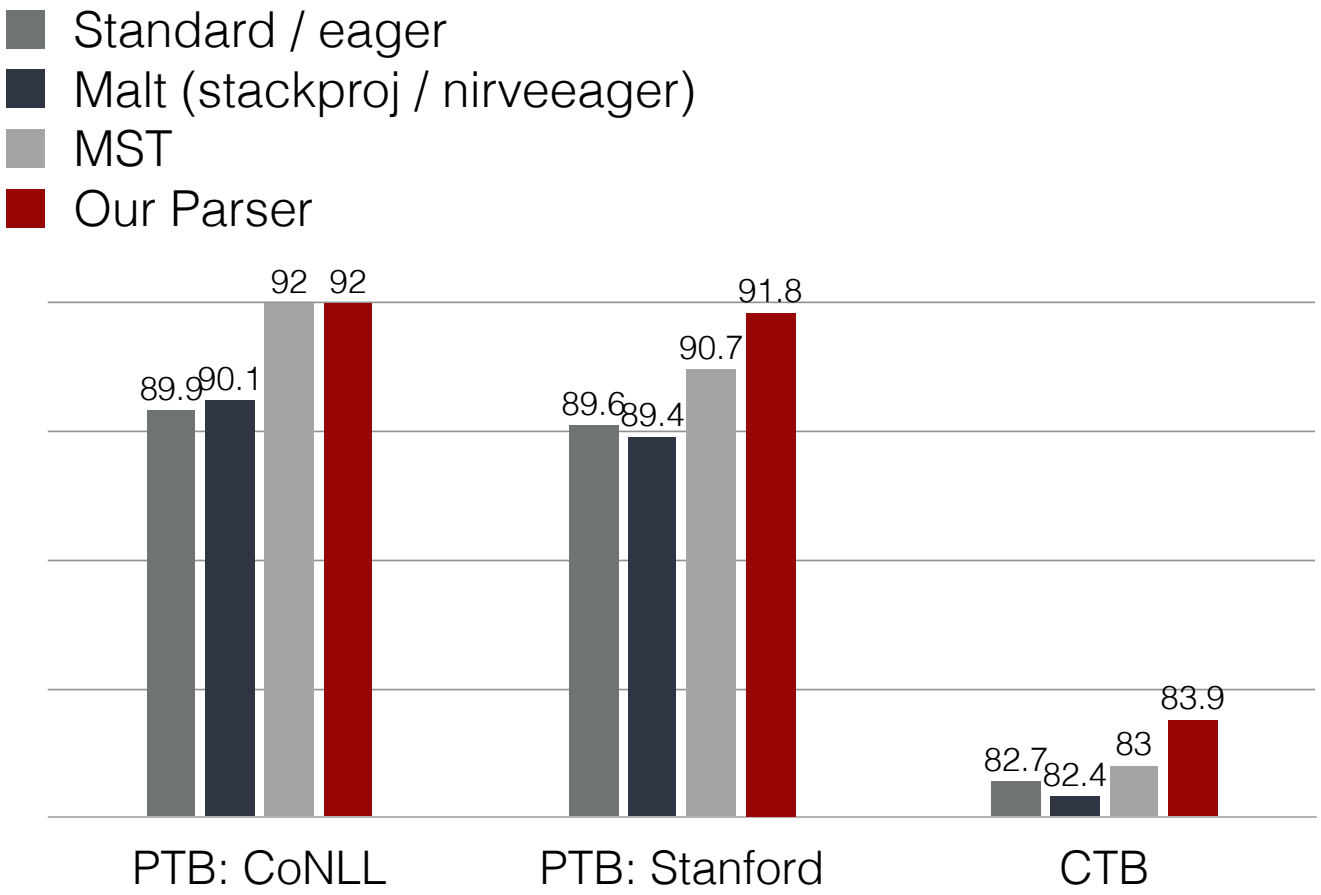
(Nivre et al 2006, McDonald and Pereira 2006)

Baselines

- **Standard / eager**: our own implemented perceptron-based greedy parsers using arc-standard or arc-eager system, with a rich feature set from (Zhang and Nivre, 2011).
- **MaltParser**
  - two algorithms **stackproj** and **nivreeager**.
- **MSTParser**

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Unlabeled Attachment Score (UAS)

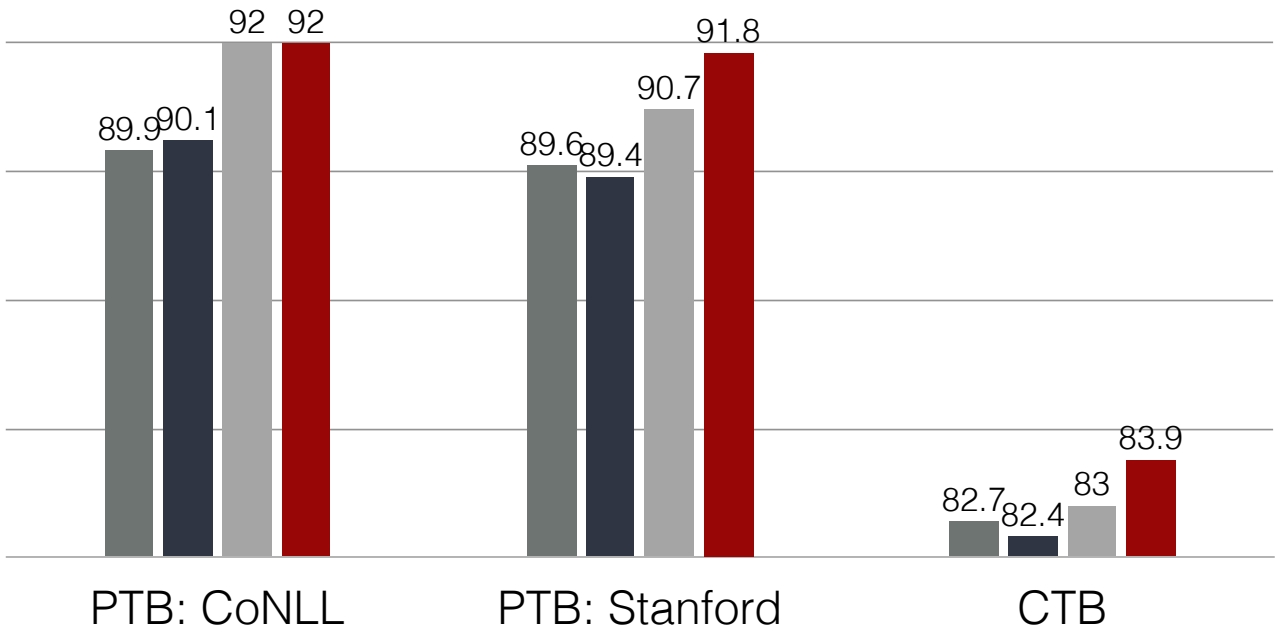


Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Unlabeled Attachment Score (UAS)

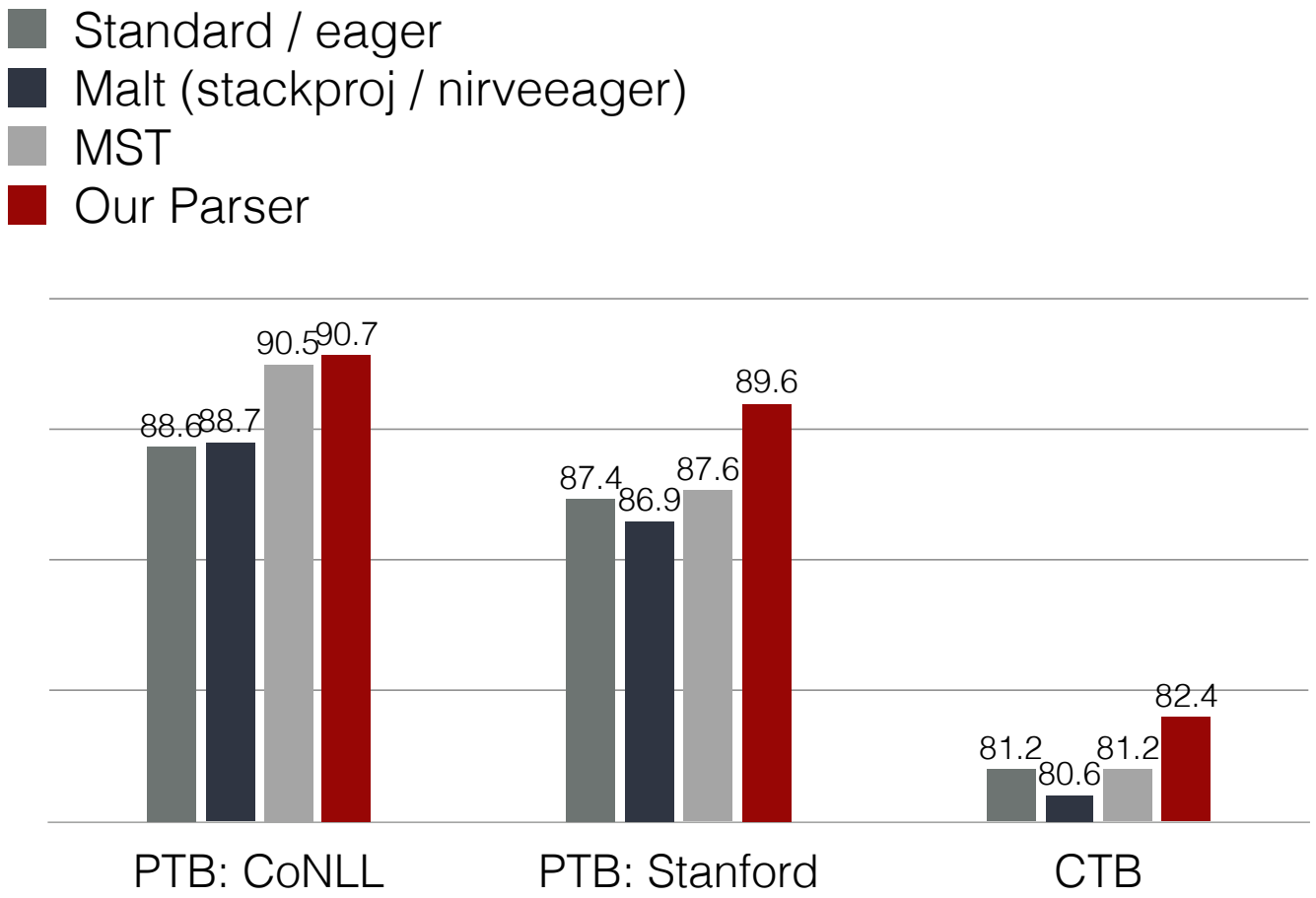
- Standard / eager
- Malt (stackproj / nirveeager)
- MST
- Our Parser

Compared with greedy parsers,
PTB: > 2.0%
CTB: >1.2%



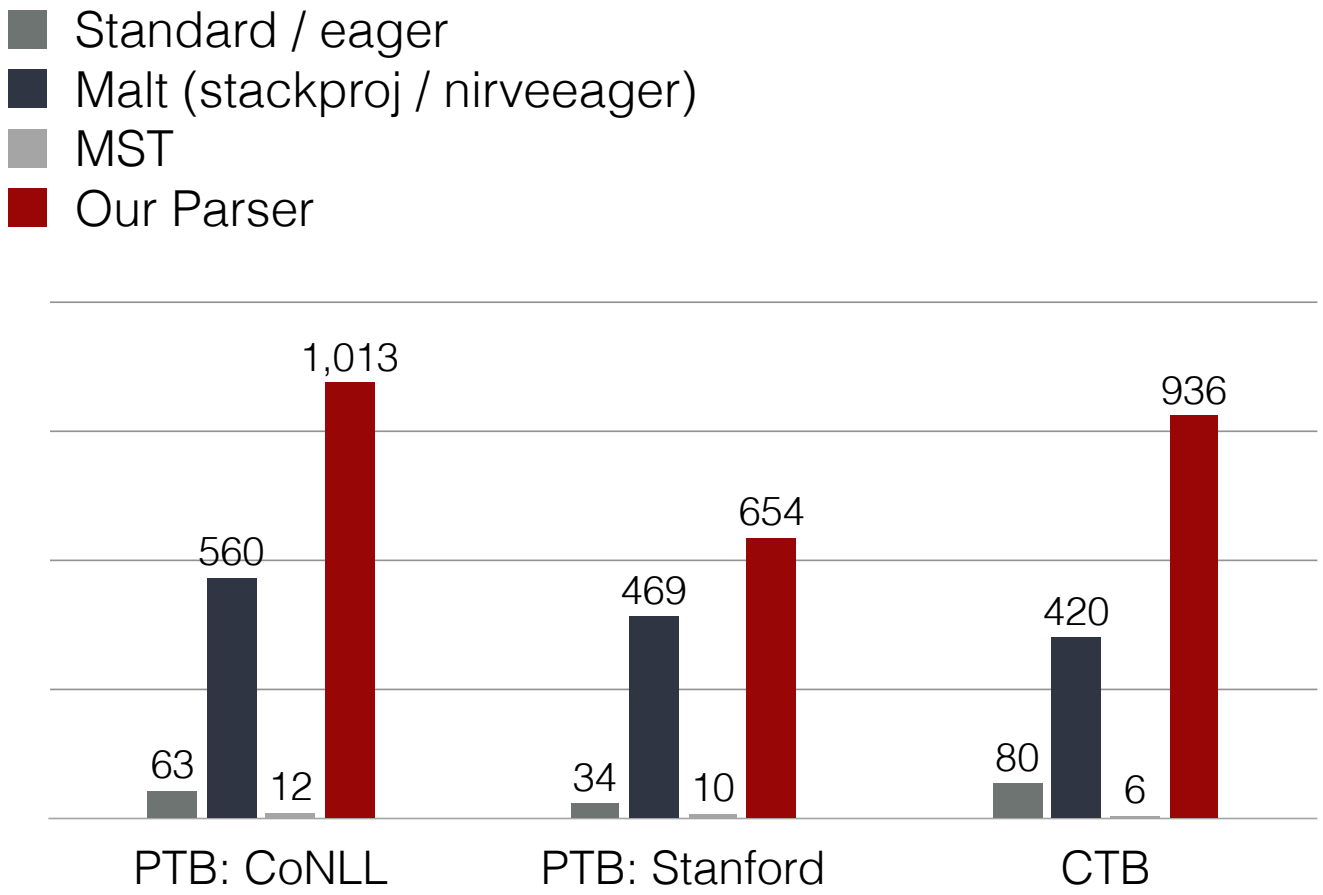
Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Labeled Attachment Score (LAS)



Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Parsing Speed (sent/s)



Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

(Zhang and Nirve 2011, Martins et al 2013)

Our Work

- A neural network based dependency parser!

Parsing on English Penn Treebank (§23):

		Unlabeled attachment score (UAS)	sent / s
Transition -based	MaltParser (greedy)	89.9	560
	Our Parser (greedy)	92.0 (+2.1)	1013 (×1.8)
	Zpar: beam = 64	92.9*	29*
Graph -based	MSTParser	92.0	12
	TurboParser	93.1*	31*

Danqi Chen and Christopher Manning, A Fast and Accurate Dependency Parser using Neural Networks (Slides)

Content

1. Grammars and syntax parsing
2. Parsing with context free grammars
3. Parsing with probabilistic context free grammars
4. Dependency parsing
5. Parsing with neural networks