Syntax and Parsing

Yoav Goldberg
Bar Ilan University
Goals

• What is parsing? why do we care?
• Phrase-based (constituency) trees
  • PCFG, CKY
• Dependency trees
  • Graph parsers, transition parsers
Sentences in natural language have structure.
Linguists create Linguistic Theories for defining this structure.
The parsing problem is recovering that structure.
John ate delicious pizza with friends
John ate delicious pizza with friends
The soup, which I expected to be good, was bad
The soup, which I expected to be good, was bad.
Why Parsing?
The soup, which I expected to be good, was bad.
sentiment analysis

The soup, which I expected to be good, was bad

negative
The soup, which I expected to be **good**, was **bad**

**sentiment analysis**

**negative**
The soup, which I expected to be good, was bad.
The soup, which I expected to be good, was bad.
knowing the structure of the sentence helps sentiment analysis
German  \rightarrow  English

machine translation

über mehrere Jahre hatte niemand in dem Haus gelebt.

down

over several years, no one had lived in the house.
machine translation

German  ➔  English

no one lived in the house for several years.

über mehrere Jahre hatte niemand in dem Haus gelebt.

over several years, no one had lived in the house.
machine translation

German  ➔  English

We suggest that NMT can also benefit the incorporation of syntactic knowledge, and propose a simple method of performing string-to-tree neural machine translation. Our method is inspired by recent works in syntactic parsing, which model dependencies between the words of the translated sentence. We omit having part of speech tags as terminals, they contain the words of the translated sentence. We omit the two models.

Figure 1: Top - a lexicalized tree translation predicted by the bpe2tree model. Bottom - a translation for the same sentence from the bpe2bpe model compared to a standard NMT system, as demonstrated in Figure 2. The linearized trees we predict are different as instead of trees as sequences (S→S→T), in which a source-language sentence is translated into a target-language string, our syntax-aware system shows that it performs better than a syntax-agnostic baseline by translating into linearized, lexicalized constituent trees, as demonstrated in Figure 3, and Charniak.

The linearized trees we predict are different as instead of trees as sequences (S→S→T), in which a source-language sentence is translated into a target-language string, our syntax-aware system shows that it per-
Knowing the structure of a sentence helps to translate better
Hypernymy Extraction

Hypernymy detection and extraction

"tuvalu" is a country
"ninjaken" is a weapon
"chlamydophila" is a bacteria
experts often define *chlamydophila* as any *bacteria* that will...
experts often define \textit{chlamydomphila} as any bacteria that will ...
experts often define chlamydophila as any bacteria that will ...
Embeddings: et al., 2014), trained on Wikipedia. We tried both pre-trained GloVe word embeddings (Pennington et al., 2014) and a character-based neural representation of words (Mikolov et al., 2010). We tuned the hyper-parameters (learning rate, batch size, etc.) using a grid search. The embeddings were initialized randomly, and we updated them during training. We used PyCNN for the implementation.

Implementation Details

Term-Pair Classification

An illustration of term-pair classification. Each term-pair is represented by several paths. Each path is a sequence of edges, and each edge consists of four components: lemma, POS, dependency label and dependency direction. Each edge vector is fed in sequence into the LSTM, resulting in a path embedding vector. The path vector is then averaged to represent the term-pair. The averaged path vector becomes the term-pair's feature vector, used for classification. The dashed lines represent the network architecture.

To train the network, we minimize the cross entropy loss using gradient-based optimization, with the supervision given for the term pairs. We represent each term-pair as the weighted-average of its path vectors, where each path vector is defined in equation 1. This way, each term-pair is represented by the multiset of lexico-syntactic features of the path vectors.

The network presented in Section 3.1 classifies term-pairs, by applying average pooling on its path vectors. We initialized the lemma embeddings with the experts' opinion. We extended the network to take into account higher-dimensional embeddings, respectively, and concatenate distributional information on each term. In this way, we considered complementary, we present a simple way to integrate distributional features in the network, yielding improved performance. To train the network, we used PyCNN. We minimized the cross entropy loss using gradient-based optimization, with the supervision given for the term pairs. We represent each term-pair as the weighted-average of its path vectors, where each path vector is defined in equation 1. This way, each term-pair is represented by the multiset of lexico-syntactic features of the path vectors. We initialized the lemma embeddings with the experts' opinion. We extended the network to take into account higher-dimensional embeddings, respectively, and concatenate distributional information on each term. In this way, we considered complementary, we present a simple way to integrate distributional features in the network, yielding improved performance. To train the network, we used PyCNN. We minimized the cross entropy loss using gradient-based optimization, with the supervision given for the term pairs. We represent each term-pair as the weighted-average of its path vectors, where each path vector is defined in equation 1. This way, each term-pair is represented by the multiset of lexico-syntactic features of the path vectors.
Embeddings:
- Lemma
- POS
- Dependency label
- Direction

The network presented in Section 3.1 classifies term-pairs as positive if their hypernymy is verified. We initialize the lemma embeddings with the pre-trained GloVe word embeddings (Pennington et al., 2014), trained on Wikipedia. We tried both pre-trained GloVe word embeddings (Pennington et al., 2014) and our own embeddings, as well as out-of-vocabulary lemmas, in the corpus. Our goal was to improve the performance on the validation set.

We minimized the cross-entropy loss using gradient-based optimization, with the initial learning rate and dropout rate tuned on the validation set (see the appendix for the hyper-parameters values). We used the PyCNN implementation to learn the features.

Path LSTM

encode tree path as LSTM
Embeddings:
- lemma
- POS
- dependency label
- direction

Pre-trained GloVe word embeddings (Pennington appendix for the hyper-parameters values).

We tuned the hyper-parameters (learning by a dropout on each of the components' embeddings). Regularization is applied in mini-batches of size 10 and the Adam update rule minimizes the cross entropy loss using gradient-based optimization, with $\text{softmax}$.

### Implementation Details

Whether work that performs binary classification to decide whether:

We then feed this path vector to a single-layer network, as follows:

- $\tilde{\phi}_p$ average pooling
- $(x, y)$ classification (softmax)

Figure 2:

An illustration of term-pair classification. Each term-pair is represented by several paths. Each path is a sequence of edges, and each edge consists of four components: lemma, POS, dependency label and dependency direction. Each edge vector is a 2-dimensional vector whose components are initialized randomly.

To train the network, we extended the network to take into account term-pair classification. The dashed term-pair as the weighted-average of its path vectors is a hypernym of $x, y$.

We initialize the lemma embeddings with the given for the term pairs. We represent each path vector defined in equation 1. This way, each feature vector, redefining the network indeed outperforms them. Yet, we consider complementary, we present a simple way to integrate distributional features in the network, respectively, and noted as $p, \ldots$.

The network presented in Section 3.1 classifies according to the paths that connected the term-pair based on the paths that connected the $50$-dimensional and $100$-dimensional embeddings, as well as out-of-vocabulary lemmas, considering complementary, we present a simple way to integrate distributional features in the network, respectively, and noted as $p, \ldots$.

Path LSTM

Term-pair Classifier

encode tree path as LSTM
based on parse-tree paths

based on distributional semantics
(which can also be aided by parsing)
based on parse-tree paths

based on distributional semantics
(which can also be aided by parsing)

substantially improves results on many lexical relation extraction tasks and datasets
Preposition Sense Disambiguation

I met him **for** lunch
He paid **for** me
We sat there **for** hours

---

Preposition sense disambiguation
+ semisup on multilingual data
Preposition Sense Disambiguation

I met him **for** lunch  
He paid **for** me  
We sat there **for** hours  

**preposition sense disambiguation**  
+ **semisup on multilingual data**
The features and the model:

The features are similar to those used in previous works. Features are extracted from:
- 2-words-window
- Head and modifier of the preposition
parse-trees play a central role in many applications.
Nice property of syntax – no need to understand

Can recover structure without understanding the words

“the plumpets and goomps ghoked the kolp”

- there is something called a plumpet. more than 1.
- there is a similar thing called a goomp. more than 1.
- plumpets and goomps are together.
- there is an action called ghoking.
- ...
Automatic parsers get it right too
Parsing is structured prediction
Natural Language Parsing

Previously

- Structure is a sequence.
- Each item can be tagged.
- We can mark some spans.
Previously

- Structure is a sequence.
- Each item can be tagged.
- We can mark some spans.

Today

- Hierarchical Structure.
Hierarchical Structure?
Structure
Example 1: math

3*2+5*3
Structure

Example 1: math

$$3 \times 2 + 5 \times 3$$
Example 1: math

$3 \times 2 + 5 \times 3$

$3 \times 2 + 5 \times 3$

$3 \times 2 + 5 \times 3$

$3 \times 2 + 5 \times 3$

$3 \times 2 + 5 \times 3$

$3 \times 2 + 5 \times 3$

$3 \times 2 + 5 \times 3$
Structure
Example 2: Language Data

Fruit flies like a banana
Structure
Example 2: Language Data

Fruit flies like a banana

Constituency Structure

S
   NP
      Adj  Noun
         Fruit  Flies
   VP
      Vb
      like
   NP
      Det  Noun
         a  banana

Dependency Structure

like
   flies  banana
   Fruit  a
In this part we concentrate on Constituency Parsing: mapping from sentences to trees with labeled nodes and the sentence words at the leaves.
Why is parsing hard?

Ambiguity

Fat people eat candy
Why is parsing hard?

Ambiguity

Fat people eat candy

S
  NP
    Adj    Nn
      Fat    people
  VP
    Vb    NP
      eat    Nn
candy
Why is parsing hard?

Ambiguity

Fat people eat candy

Fat people eat accumulates

S
  NP     VP
   Adj  Nn  Vb  NP
     Fat  people  eat  Nn
               candy
Why is parsing hard?

Ambiguity

Fat people eat candy

Fat people eat accumulates
Why is parsing hard?

Ambiguity

- I ate pizza with anchovies
- I ate pizza with friends
Why is parsing hard?
Real Sentences are long…

“Former Beatle Paul McCartney today was ordered to pay nearly $50M to his estranged wife as their bitter divorce battle came to an end.”

“Welcome to our Columbus hotels guide, where you’ll find honest, concise hotel reviews, all discounts, a lowest rate guarantee, and no booking fees.”
Let’s learn how to parse
Let’s learn how to parse . . . but first lets review some stuff from Automata and Formal Languages.
A context free grammar $G = (N, \Sigma, R, S)$ where:

- $N$ is a set of non-terminal symbols
- $\Sigma$ is a set of terminal symbols
- $R$ is a set of rules of the form $X \rightarrow Y_1 Y_2 \cdots Y_n$ for $n \geq 0$, $X \in N$, $Y_i \in (N \cup \Sigma)$
- $S \in N$ is a special start symbol
Context Free Grammars

a simple grammar

\[ N = \{ S, NP, VP, Adj, Det, Vb, Noun \} \]
\[ \Sigma = \{ fruit, flies, like, a, banana, tomato, angry \} \]
\[ S = 'S' \]
\[ R = \]

\[ S \rightarrow NP \ VP \]
\[ NP \rightarrow Adj \ Noun \]
\[ NP \rightarrow Det \ Noun \]
\[ VP \rightarrow Vb \ NP \]
\[ Adj \rightarrow fruit \]
\[ Noun \rightarrow flies \]
\[ Vb \rightarrow like \]
\[ Det \rightarrow a \]
\[ Noun \rightarrow banana \]
\[ Noun \rightarrow tomato \]
\[ Adj \rightarrow angry \]
Left-most derivations

Left-most derivation is a sequence of strings $s_1, \cdots, s_n$ where

- $s_1 = S$ the start symbol
- $s_n \in \Sigma^*$, meaning $s_n$ is only terminal symbols
- Each $s_i$ for $i = 2 \cdots n$ is derived from $s_{i-1}$ by picking the left-most non-terminal $X$ in $s_{i-1}$ and replacing it by some $\beta$ where $X \rightarrow \beta$ is a rule in $R$. 

"For example: [S], [NP VP], [Adj Noun VP], [fruit Noun VP], [fruit flies VP], [fruit flies Vb NP], [fruit flies like NP], [fruit flies like Det Noun], [fruit flies like a], [fruit flies like a banana]."
Left-most derivations

Left-most derivation is a sequence of strings \( s_1, \ldots, s_n \) where

- \( s_1 = S \) the start symbol
- \( s_n \in \Sigma^* \), meaning \( s_n \) is only terminal symbols
- Each \( s_i \) for \( i = 2 \cdots n \) is derived from \( s_{i-1} \) by picking the left-most non-terminal \( X \) in \( s_{i-1} \) and replacing it by some \( \beta \) where \( X \rightarrow \beta \) is a rule in \( R \).

For example: \( [S],[NP \ VP],[Adj \ Noun \ VP], \ [fruit \ Noun \ VP], \ [fruit \ flies \ VP], [fruit \ flies \ Vb \ NP], [fruit \ flies \ like \ NP], \ [fruit \ flies \ like \ Det \ Noun], \ [fruit \ flies \ like \ a], \ [fruit \ flies \ like \ a \ banana] \)
Left-most derivation example

S
Left-most derivation example

S → NP VP

The resulting derivation can be written as a tree.

Many trees can be generated.
Left-most derivation example

S
NP VP
Adj Noun VP

NP → Adj Noun

The resulting derivation can be written as a tree.

Many trees can be generated.
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP

Adj → fruit
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP

Noun → flies
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP

VP → Vb NP
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP
fruit flies like NP

Vb → like
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP
fruit flies like NP
fruit flies like Det Noun

NP → Det Noun
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP
fruit flies like NP
fruit flies like Det Noun
fruit flies like a Noun

Det → a
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP
fruit flies like NP
fruit flies like Det Noun
fruit flies like a Noun
fruit flies like a banana

Noun → banana
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP
fruit flies like NP
fruit flies like Det Noun
fruit flies like a Noun
fruit flies like a banana

- The resulting derivation can be written as a tree.
Left-most derivation example

S
NP VP
Adj Noun VP
fruit Noun VP
fruit flies VP
fruit flies Vb NP
fruit flies like NP
fruit flies like Det Noun
fruit flies like a Noun
fruit flies like a banana

- The resulting derivation can be written as a tree.
- Many trees can be generated.
Context Free Grammars

a simple grammar

S → NP VP
NP → Adj Noun
NP → Det Noun
VP → Vb NP

Adj → fruit
Noun → flies
Vb → like
Det → a
Noun → banana
Noun → tomato
Adj → angry

...
a simple grammar

S → NP VP
NP → Adj Noun
NP → Det Noun
VP → Vb NP

Example

S
  |   |
  NP    VP
  |   |
  Adj  Noun  Vb  NP
  |   |
  Fruit Flies like a Noun
  |   |
  a banana

...
Context Free Grammars

a simple grammar

S → NP VP
NP → Adj Noun
NP → Det Noun
VP → Vb NP

Adj → fruit
Noun → flies
Vb → like
Det → a
Noun → banana
Noun → tomato
Adj → angry

Example

S

<table>
<thead>
<tr>
<th>NP</th>
<th>VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adj</td>
<td>Noun</td>
</tr>
<tr>
<td>Angry</td>
<td>Flies</td>
</tr>
<tr>
<td>Vb</td>
<td>Det</td>
</tr>
<tr>
<td>like</td>
<td>a</td>
</tr>
</tbody>
</table>
a simple grammar

S → NP VP
NP → Adj Noun
NP → Det Noun
VP → Vb NP

- 
  Adj → fruit
  Noun → flies
  Vb → like
  Det → a
  Noun → banana
  Noun → tomato
  Adj → angry

...
a simple grammar

S → NP VP
NP → Adj Noun
NP → Det Noun
VP → Vb NP

- Adj → fruit
Noun → flies
Vb → like
Det → a
Noun → banana
Noun → tomato
Adj → angry

...
a simple grammar

S → NP VP
NP → Adj Noun
NP → Det Noun
VP → Vb NP

- Adj → fruit
Noun → flies
Vb → like
Det → a
Noun → banana
Noun → tomato
Adj → angry

Example

S
  / \  
NP   VP
  /   /
Det Noun Vb NP
    /  /
  a banana like
    /  /
  a tomato
a simple grammar

S → NP VP
NP → Adj Noun
NP → Det Noun
VP → Vb NP

- Adj → fruit
Noun → flies
Vb → like
Det → a
Noun → banana
Noun → tomato
Adj → angry

Example

```
S
  /\   /
 NP VP
  /\   /
 Det Noun Vb NP
     /\   /
    a banana like Adj banana
         /\   /
        angry tomato
         /
        "banana"
```
The parsing problem

Given a string, recover the derivation.
Parsing with (P)CFGs
Let’s assume... 

- Let’s assume natural language is generated by a CFG.
- ...and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from \( S \) that generates it.
Parsing with CFGs

Let’s assume…

- Let’s assume natural language is generated by a CFG.
- . . . and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from $S$ that generates it.

Problem

- Natural Language is NOT generated by a CFG.
  - We can find $a^n b^n c^n$ structures, and many other arguments.
Parsing with CFGs

Let’s assume...

- Let’s assume natural language is generated by a CFG.
- ... and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from $S$ that generates it.

Problem

- Natural Language is NOT generated by a CFG.
  - We can find $a^n b^n c^n$ structures, and many other arguments.

Solution

- We assume really hard that it is.
Parsing with CFGs

Let’s assume...

- Let’s assume natural language is generated by a CFG.
- ... and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from $S$ that generates it.

Problem

- We don’t have the grammar.
Parsing with CFGs

Let’s assume...

- Let’s assume natural language is generated by a CFG.
- ...and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from $S$ that generates it.

Problem

- We don’t have the grammar.

Solution

- We’ll ask a genius linguist to write it!
Parsing with CFGs

Let’s assume...

- Let’s assume natural language is generated by a CFG.
- ...and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from $S$ that generates it.

Problem

- How do we find the chain of derivations?
Parsing with CFGs

Let’s assume...

- Let’s assume natural language is generated by a CFG.
- ...and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from S that generates it.

Problem

- How do we find the chain of derivations?

Solution

- With dynamic programming! (soon)
Parsing with CFGs

Let’s assume...

- Let’s assume natural language is generated by a CFG.
- ...and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from $S$ that generates it.

Problem

- Real grammar: hundreds of possible derivations per sentence.
Parsing with CFGs

Let’s assume...

- Let’s assume natural language is generated by a CFG.
- . . . and let’s assume we have the grammar.
- Then parsing is easy: given a sentence, find the chain of derivations starting from $S$ that generates it.

Problem

- Real grammar: hundreds of possible derivations per sentence.

Solution

- No problem! We’ll choose the best one. (sooner)
Obtaining a Grammar

Let a genius linguist write it

- Hard. Many rules, many complex interactions.
- Genius linguists don’t grow on trees!
Obtaining a Grammar

Let a genius linguist write it

- Hard. Many rules, many complex interactions.
- Genius linguists don’t grow on trees!

An easier way: ask a linguist to grow trees

- Ask a linguist to annotate sentences with tree structure.
- (This need not be a genius—Smart is enough.)
- Then extract the rules from the annotated trees.

Treebanks

- English Treebank: 40k sentences, manually annotated
- Other languages: often about 5k sentences.
Obtaining a Grammar

Let a genius linguist write it

- Hard. Many rules, many complex interactions.
- Genius linguists don’t grow on trees!

An easier way: ask a linguist to grow trees

- Ask a linguist to annotate sentences with tree structure.
- (This need not be a genius—Smart is enough.)
- Then extract the rules from the annotated trees.

Treebanks

- **English Treebank**: 40k sentences, manually annotated with tree structure.
- **Other languages**: often about 5k sentences.
Treebank Sentence Example

(S
  (NP-SBJ
    (NP (NNP Pierre) (NNP Vinken) )
    (, ,)
    (ADJP
      (NP (CD 61) (NNS years) )
      (JJ old) )
    (, ,) )
  (VP (MD will)
    (VP (VB join)
      (NP (DT the) (NN board) )
      (PP-CLR (IN as)
        (NP (DT a) (JJ nonexecutive) (NN director)
         (NP-TMP (NNP Nov.) (CD 29) ))
      (, ) ))
  (, ,) )
)
Supervised Learning from a Treebank
Extracting CFG from Trees

- The leafs of the trees define $\Sigma$
- The internal nodes of the trees define $N$
- Add a special $S$ symbol on top of all trees
- Each node an its children is a rule in $R$
Extracting CFG from Trees

- The leaves of the trees define $\Sigma$
- The internal nodes of the trees define $N$
- Add a special $S$ symbol on top of all trees
- Each node and its children is a rule in $R$

Extracting Rules

```
S
   /\   \\
  NP   VP
  /\   /\  \\
Adj Noun Vb NP
Fruit Flies like Det Noun
        \\
a  banana
```
Extracting CFG from Trees

- The leaves of the trees define $\Sigma$
- The internal nodes of the trees define $N$
- Add a special $S$ symbol on top of all trees
- Each node and its children is a rule in $R$

Extracting Rules

$S \rightarrow NP \; VP$

S

NP

Adj

Fruit

Noun

Flies

VP

Vb

like

Det

Noun

a

banana
Extracting CFG from Trees

- The leafs of the trees define $\Sigma$
- The internal nodes of the trees define $N$
- Add a special $S$ symbol on top of all trees
- Each node and its children is a rule in $R$

Extracting Rules

$$S \rightarrow NP \ VP$$

$$NP \rightarrow Adj \ Noun$$

$$VP \rightarrow Vb \ NP$$

- Fruit
- Flies
- like
- a
- banana
Extracting CFG from Trees

- The leafs of the trees define $\Sigma$
- The internal nodes of the trees define $N$
- Add a special $S$ symbol on top of all trees
- Each node an its children is a rule in $R$

Extracting Rules

```
S
  NP   VP
    Adj Noun  Vb NP
        Fruit  Flies like Det Noun
```

$S \rightarrow NP\ VP$
$NP \rightarrow Adj\ Noun$
$Adj \rightarrow fruit$
From CFG to PCFG

- English is NOT generated from CFG ⇒ It’s generated by a PCFG!
From CFG to PCFG

- English is NOT generated from CFG ⇒ It’s generated by a PCFG!
- PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- All probabilities for the same LHS sum to 1.
From CFG to PCFG

- English is NOT generated from CFG ⇒ It’s generated by a PCFG!
- PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- All probabilities for the same LHS sum to 1.
- Multiplying all the rule probs in a derivation gives the probability of the derivation.
- We want the tree with maximum probability.
From CFG to PCFG

- English is NOT generated from CFG \(\Rightarrow\) It’s generated by a PCFG!
- PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- All probabilities for the same LHS sum to 1.
- Multiplying all the rule probs in a derivation gives the probability of the derivation.
- We want the tree with maximum probability.

More Formally

\[
P(\text{tree, sent}) = \prod_{l \rightarrow r \in \text{deriv(tree)}} p(l \rightarrow r)
\]
From CFG to PCFG

- English is NOT generated from CFG ⇒ It’s generated by a PCFG!
- PCFG: probabilistic context free grammar. Just like a CFG, but each rule has an associated probability.
- All probabilities for the same LHS sum to 1.
- Multiplying all the rule probs in a derivation gives the probability of the derivation.
- We want the tree with maximum probability.

More Formally

\[ P(tree, sent) = \prod_{l \rightarrow r \in \text{deriv}(tree)} p(l \rightarrow r) \]

\[ tree = \arg \max_{tree \in \text{trees}(sent)} P(tree|sent) = \arg \max_{tree \in \text{trees}(sent)} P(tree, sent) \]
Just structure prediction, really

\[
\text{score}(\text{tree}) = \prod_{l \rightarrow r \in \text{deriv}(\text{tree})} p(l \rightarrow r)
\]

\[
\text{score}(\text{tree}) = \sum_{l \rightarrow r \in \text{deriv}(\text{tree})} \log p(l \rightarrow r)
\]

\[
\text{score}(\text{tree}) = \sum_{l \rightarrow r \in \text{deriv}(\text{tree})} \text{score}(l \rightarrow r)
\]
PCFG Example

a simple PCFG

1.0  S  →  NP  VP
0.3  NP  →  Adj  Noun
0.7  NP  →  Det  Noun
1.0  VP  →  Vb  NP

-  
0.2  Adj  →  fruit
0.2  Noun  →  flies
1.0  Vb  →  like
1.0  Det  →  a
0.4  Noun  →  banana
0.4  Noun  →  tomato
0.8  Adj  →  angry

Example

S
  NP  VP
  |  |
  Adj  Noun  Vb  NP
  |  |
  Fruit  Flies  like  Det  Noun
  |  |
  a  banana

1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033
PCFG Example

a simple PCFG
1.0 $S \rightarrow NP \ VP$
0.3 $NP \rightarrow Adj \ Noun$
0.7 $NP \rightarrow Det \ Noun$
1.0 $VP \rightarrow Vb \ NP$

- 
0.2 $Adj \rightarrow \text{fruit}$
0.2 $Noun \rightarrow \text{flies}$
1.0 $Vb \rightarrow \text{like}$
1.0 $Det \rightarrow a$
0.4 $Noun \rightarrow \text{banana}$
0.4 $Noun \rightarrow \text{tomato}$
0.8 $Adj \rightarrow \text{angry}$

Example

$$1 \times 0.3 \times 0.2 \times 0.7 \times 1.0 \times 0.2 \times 1 \times 1 \times 0.4 = 0.0033$$
a simple PCFG

1.0 $S \rightarrow NP \ VP$
0.3 $NP \rightarrow Adj \ Noun$
0.7 $NP \rightarrow Det \ Noun$
1.0 $VP \rightarrow Vb \ NP$

-  
0.2 $Adj \rightarrow fruit$
0.2 $Noun \rightarrow flies$
1.0 $Vb \rightarrow like$
1.0 $Det \rightarrow a$
0.4 $Noun \rightarrow banana$
0.4 $Noun \rightarrow tomato$
0.8 $Adj \rightarrow angry$

Example

$S$

$NP$

$Adj$

$Noun$

$Flies$

$like$

$Det$

$NP$

$a$

$banana$

$1 \times 0.3 \times 0.2 \times 0.7 \times 1.0 \times 0.2 \times 1 \times 1 \times 0.4 = 0.0033$
a simple PCFG

1.0 S → NP VP
0.3 NP → Adj Noun
0.7 NP → Det Noun
1.0 VP → Vb NP

- 0.2 Adj → fruit
0.2 Noun → flies
1.0 Vb → like
1.0 Det → a
0.4 Noun → banana
0.4 Noun → tomato
0.8 Adj → angry

Example

S
  NP
    Adj
    Fruit
  Noun
  VP
    Vb
    like
    Det
    a
    Noun
    banana

1 * 0.3 * 0.2 * 0.7 * 1.0 * 0.2 * 1 * 1 * 0.4 = 0.0033
Parsing with PCFG

- Parsing with a PCFG is finding the most probable derivation for a given sentence.
- This can be done quite efficiently with dynamic programming (the CKY algorithm)
Parsing with PCFG

- Parsing with a PCFG is finding the most probable derivation for a given sentence.
- This can be done quite efficiently with dynamic programming (the CKY algorithm)

Obtaining the probabilities

- We estimate them from the Treebank.
- \[ P(LHS \rightarrow RHS) = \frac{\text{count}(LHS \rightarrow RHS)}{\text{count}(LHS \rightarrow \cdot)} \]
- We can also add smoothing and backoff, as before.
- Dealing with unknown words - like in the HMM
The CKY algorithm
The Problem

Input

- Sentence (a list of words)
  - $n$ – sentence length
- CFG Grammar (with weights on rules)
  - $g$ – number of non-terminal symbols

Output

- A parse tree / the best parse tree

But... 

- Exponentially many possible parse trees!

Solution

- Dynamic Programming!
CKY

Cocke    Kasami    Younger
Cocke  Kasami  Younger
196?
CKY

Cocke    Kasami    Younger
196?    1965
<table>
<thead>
<tr>
<th>Cocke</th>
<th>Kasami</th>
<th>Younger</th>
</tr>
</thead>
<tbody>
<tr>
<td>196?</td>
<td>1965</td>
<td>1967</td>
</tr>
</tbody>
</table>
3 Interesting Problems

- Recognition
- Parsing
- Disambiguation
3 Interesting Problems

- Recognition
  - Can this string be generated by the grammar?
- Parsing
- Disambiguation
3 Interesting Problems

- Recognition
  - Can this string be generated by the grammar?
- Parsing
  - Show me a possible derivation...
- Disambiguation
3 Interesting Problems

- Recognition
  - Can this string be generated by the grammar?
- Parsing
  - Show me a possible derivation...
- Disambiguation
  - Show me THE BEST derivation
3 Interesting Problems

- Recognition
  - Can this string be generated by the grammar?
- Parsing
  - Show me a possible derivation...
- Disambiguation
  - Show me THE BEST derivation

CKY can do all of these in polynomial time
3 Interesting Problems

- Recognition
  - Can this string be generated by the grammar?
- Parsing
  - Show me a possible derivation...
- Disambiguation
  - Show me THE BEST derivation

CKY can do all of these in polynomial time

- For any CNF grammar
Definition
A CFG is in CNF form if it only has rules like:

- $A \rightarrow B \ C$
- $A \rightarrow \alpha$

$A,B,C$ are non terminal symbols
$\alpha$ is a terminal symbol (a word...)

- All terminal symbols are RHS of unary rules
- All non terminal symbols are RHS of binary rules

CKY can be easily extended to handle also unary rules: $A \rightarrow B$
Binarization

Fact

- Any CFG grammar can be converted to CNF form
Binarization

Fact

- Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- We already have $A \rightarrow \alpha$
Binarization

Fact

- Any CFG grammar can be converted to CNF form

Specifcally for Natural Language grammars

- We already have $A \rightarrow \alpha$
  - $(A \rightarrow \alpha \beta$ is also easy to handle)
Binarization

Fact

- Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- We already have $A \rightarrow \alpha$
  - $(A \rightarrow \alpha \beta$ is also easy to handle)
- Unary rules $(A \rightarrow B)$ are OK
Binarization

Fact

- Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- We already have $A \rightarrow \alpha$
  - $(A \rightarrow \alpha \beta$ is also easy to handle)
- Unary rules $(A \rightarrow B)$ are OK
- Only problem: $S \rightarrow NP \ PP \ VP \ PP$
Binarization

Fact

- Any CFG grammar can be converted to CNF form

Specifically for Natural Language grammars

- We already have $A \rightarrow \alpha$
  - $(A \rightarrow \alpha \beta$ is also easy to handle)
- Unary rules $(A \rightarrow B)$ are OK
- Only problem: $S \rightarrow NP\, PP\, VP\, PP$

Binarization

\[
\begin{align*}
S & \rightarrow NP\, NP\_PP\_VP\_PP \\
NP\_PP\_VP\_PP & \rightarrow PP\, NP\_PP\_VP\_PP \\
NP\_PP\_VP\_PP & \rightarrow VP\, NP\_PP\_VP\_PP
\end{align*}
\]
Finally, CKY

**Recognition**

- **Main idea:**
  - Build parse tree from bottom up
  - Combine built trees to form bigger trees using grammar rules
  - When left with a single tree, verify root is $S$

- Exponentially many possible trees...
  - Search over all of them in polynomial time using DP
  - Shared structure – smaller trees
Main Idea

If we know:

- $w_i \ldots w_j$ is an NP
- $w_{j+1} \ldots w_k$ is a VP

and grammar has rule:

- $S \rightarrow NP \ VP$

Then we know:

- $S$ can derive $w_i \ldots w_k$
If we know:

- $w_i \ldots w_j$ is an $NP$
- $w_{j+1} \ldots w_k$ is a $VP$

and grammar has rule:

- $S \rightarrow NP \ VP$

Then we know:

- $S$ can derive $w_i \ldots w_k$
Main Idea

If we know:

- \( w_i \ldots w_j \) is an NP
- \( w_{j+1} \ldots w_k \) is a VP

and grammar has rule:

- \( S \rightarrow NP \ VP \)

Then we know:

- \( S \) can derive \( w_i \ldots w_k \)
If we know:

- \( w_i \ldots w_j \) is an NP
- \( w_{j+1} \ldots w_k \) is a VP

and grammar has rule:

- \( S \rightarrow NP \ VP \)

Then we know:

- \( S \) can derive \( w_i \ldots w_k \)
Data Structure

(Half a) two dimensional array \((n \times n)\)
Data Structure

On its side
Data Structure

Each cell: all nonterminals than can derive word $i$ to word $j$
Data Structure

Each cell: all nonterminals than can derive word $i$ to word $j$
imagine each cell as a $g$ dimensional array
Data Structure

Each cell: all nonterminals than can derive word $i$ to word $j$
imagine each cell as a $g$ dimensional array

```
S   x
NP  x
VP  v
PP  x
AdjP x
```

```
S   v
NP  v
VP  v
PP  x
AdjP x
```

Sue  saw  her  girl  with  a  telescope
Filling the table

<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>saw</td>
<td>her</td>
<td>girl</td>
<td>with</td>
<td>a</td>
<td>telescope</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>-----------</td>
</tr>
</tbody>
</table>

Filling the table
<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
</table>

Filling the table
Filling the table

Sue saw her girl with a telescope.
Sue saw her girl with a telescope
Filling the table

<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
</table>
Sue saw her girl with a telescope.
<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
</table>
Filling the table

<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
</table>

Sue saw her girl with a telescope.
Filling the table

Sue saw her girl with a telescope
Filling the table

Sue saw her girl with a telescope
Filling the table

<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
</table>
Filling the table

<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
</table>

Sue saw her girl with a telescope.
Sue saw her girl with a telescope
Filling the table

<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
</table>

Sue saw her girl with a telescope.
Sue saw her girl with a telescope
Filling the table

<table>
<thead>
<tr>
<th>Sue</th>
<th>saw</th>
<th>her</th>
<th>girl</th>
<th>with</th>
<th>a</th>
<th>telescope</th>
</tr>
</thead>
</table>

Sue saw her girl with a telescope.
Handling Unary rules?

Sue saw her girl with a telescope
Which Order?

Sue saw her boy with a telescope
When handling a cell, we need its potential parts to be complete.
Which Order?

Sue saw her boy with a telescope
Which Order?

Sue saw her boy with a telescope
Which Order?

Sue saw her boy with a telescope
Sue saw her boy with a telescope.
Which Order?

Sue saw her boy with a telescope
Which Order?

Sue saw her boy with a telescope.
Which Order?

Sue saw her boy with a telescope
Which Order?

Sue saw her boy with a telescope
Which Order?

Sue saw her boy with a telescope
Which Order?

"left corner parsing"

Sue saw her boy with a telescope
Complexity?
Complexity?

- $n^2 g$ cells to fill
Complexity?

- $n^2 g$ cells to fill
- $g^2 n$ ways to fill each one
Complexity?

- $n^2 g$ cells to fill
- $g^2 n$ ways to fill each one

$O(g^3 n^3)$
Finding a parse

**Parsing** – we want to actually find a parse tree

Easy: also keep a possible split point for each NT
PCFG Parsing and Disambiguation

Disambiguation – we want THE BEST parse tree

Easy: for each NT, keep best split point, and score.
PCFG Parsing Recap

- Extract grammar + probabilities from treebank.
- Given a sentence, use CKY to recover to highest scoring tree.
Doesn't really work

• It recovers the best tree according to the grammar.
• But these best trees are quite bad.
• ~73 F1 score.
Some limitations of PCFGs

• Not sensitive to words.

• Not sensitive to structural frequencies.
Another Case of PP Attachment Ambiguity

(Example from Mike Collins)
If \( q(\text{NP} \rightarrow \text{NP} \text{ PP}) > q(\text{VP} \rightarrow \text{VP PP}) \) then (b) is more probable, else (a) is more probable.

Attachment decision is completely independent of the words.

(Another Case of PP Attachment Ambiguity)

Rules

\[
\begin{align*}
S & \rightarrow \text{NP VP} \\
\text{NP} & \rightarrow \text{NNS} \\
\text{VP} & \rightarrow \text{VP PP} \\
\text{VP} & \rightarrow \text{VBD NP} \\
\text{NP} & \rightarrow \text{NNS} \\
\text{PP} & \rightarrow \text{IN NP} \\
\text{NP} & \rightarrow \text{DT NN} \\
\text{NNS} & \rightarrow \text{workers} \\
\text{VBD} & \rightarrow \text{dumped} \\
\text{NNS} & \rightarrow \text{sacks} \\
\text{IN} & \rightarrow \text{into} \\
\text{DT} & \rightarrow \text{a} \\
\text{NN} & \rightarrow \text{bin}
\end{align*}
\]

(example from Mike Collins)
Another Case of PP Attachment Ambiguity

(a) If \( q(\text{NP} \rightarrow \text{NP} \text{ PP}) > q(\text{VP} \rightarrow \text{VP} \text{ PP}) \) then (b) is more probable, else (a) is more probable.

Attachment decision is completely independent of the words!

(b) The only difference is in the attachment decision.

(Example from Mike Collins)
AC a s eo f C o o r d i n a t i o n A m b i g u i t y
(a)
NP
NP
NP
NP
NNS
dogs
IN
in
NP
houses
CC
and
NP
NNS
cats

(b)
NP
NP
NP
NP
NNS
dogs
IN
in
NP
houses
CC
and
NP
NNS
cats

(example from Mike Collins)
Here the two parses have identical rules, and therefore have identical probability under any assignment of PCFG rule probabilities.
Identical set of rules.
Same score under any assignment of PCFG rule probabilities.

(example from Mike Collins)
Structural Preferences: Close Attachment

- Example: president of a company in Africa
- Both parses have the same rules, therefore receive same probability under a PCFG
- “Close attachment” (structure (a)) is twice as likely in Wall Street Journal text.
Lexicalized PCFGs

PCFG Problem 1
Lack of sensitivity to lexical information (words)

Solution

- Make PCFG aware of words (*lexicalized* PCFG)
- Main Idea: **Head Words**
Head Words

Each constituent has one word which captures its “essence”.

hat is the “semantic head”

with is the “functional head”

(it is common to choose the functional head)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
- (VP saw the young boy with the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
- (VP saw the young boy with the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
- (VP saw the young boy with the large hat)
- (NP the young boy with the large hat)
Each constituent has one word which captures its “essence”.

- (S John *saw* the young boy with the large hat)
- (VP *saw* the young boy with the large hat)
- (NP the young *boy* with the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
- (VP saw the young boy with the large hat)
- (NP the young boy with the large hat)
- (NP the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John *saw* the young boy with the large hat)
- (VP *saw* the young boy with the large hat)
- (NP the young *boy* with the large hat)
- (NP the large *hat*)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
- (VP saw the young boy with the large hat)
- (NP the young boy with the large hat)
- (NP the large hat)
- (PP with the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)
- (NP the large **hat**)
- (PP **with** the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
- (VP saw the young boy with the large hat)
- (NP the young boy with the large hat)
- (NP the large hat)
- (PP with the large hat)
Head Words

Each constituent has one word which captures its “essence”.

- (S John saw the young boy with the large hat)
- (VP saw the young boy with the large hat)
- (NP the young boy with the large hat)
- (NP the large hat)
- (PP with the large hat)
  - hat is the “semantic head”
  - with is the “functional head”
  - (it is common to choose the functional head)
More about Heads

- Each context-free rule has one “special” child that is the head of the rule. e.g.,
  
  \[
  \begin{align*}
  S & \Rightarrow \text{NP} \quad \text{VP} \quad \text{(VP is the head)} \\
  \text{VP} & \Rightarrow \text{Vt} \quad \text{NP} \quad \text{(Vt is the head)} \\
  \text{NP} & \Rightarrow \text{DT} \quad \text{NN} \quad \text{NN} \quad \text{(NN is the head)}
  \end{align*}
  \]

- A core idea in syntax
  
  (e.g., see X-bar Theory, Head-Driven Phrase Structure Grammar)

- Some intuitions:
  - The central sub-constituent of each rule.
  - The semantic predicate in each rule.
Adding Headwords to Trees

(Slide from Mike Collins)
A constituent receives its **headword** from its **head child**.

- **S** $\Rightarrow$ **NP** **VP**  \hspace{1cm} (S receives headword from VP)
- **VP** $\Rightarrow$ **Vt** **NP**  \hspace{1cm} (VP receives headword from Vt)
- **NP** $\Rightarrow$ **DT** **NN**  \hspace{1cm} (NP receives headword from NN)

(Slide from Mike Collins)
The lawyer questioned the witness.
Chomsky Normal Form

A context free grammar $G = (N, \Sigma, R, S)$ in Chomsky Normal Form is as follows

- $N$ is a set of non-terminal symbols
- $\Sigma$ is a set of terminal symbols
- $R$ is a set of rules which take one of two forms:
  - $X \rightarrow Y_1 Y_2$ for $X \in N$, and $Y_1, Y_2 \in N$
  - $X \rightarrow Y$ for $X \in N$, and $Y \in \Sigma$
- $S \in N$ is a distinguished start symbol

We can find the highest scoring parse under a PCFG in this form, in $O(n^3|N|^3)$ time where $n$ is the length of the string being parsed.
Lexicalized Context-Free Grammars in Chomsky Normal Form

- $N$ is a set of non-terminal symbols
- $\Sigma$ is a set of terminal symbols
- $R$ is a set of rules which take one of three forms:
  - $X(h) \rightarrow_1 Y_1(h) Y_2(w)$ for $X \in N$, and $Y_1, Y_2 \in N$, and $h, w \in \Sigma$
  - $X(h) \rightarrow_2 Y_1(w) Y_2(h)$ for $X \in N$, and $Y_1, Y_2 \in N$, and $h, w \in \Sigma$
  - $X(h) \rightarrow h$ for $X \in N$, and $h \in \Sigma$
- $S \in N$ is a distinguished start symbol
An Example

S(saw) →₂ NP(man) VP(saw)
VP(saw) →₁ Vt(saw) NP(dog)
NP(man) →₂ DT(the) NN(man)
NP(dog) →₂ DT(the) NN(dog)
Vt(saw) → saw
DT(the) → the
NN(man) → man
NN(dog) → dog

(slide from Mike Collins)
Parsing with Lexicalized CFGs

- The new form of grammar looks just like a Chomsky normal form CFG, but with potentially $O(|\Sigma|^2 \times |N|^3)$ possible rules.

- Naively, parsing an $n$ word sentence using the dynamic programming algorithm will take $O(n^3|\Sigma|^2|N|^3)$ time. But $|\Sigma|$ can be huge!!

- Crucial observation: at most $O(n^2 \times |N|^3)$ rules can be applicable to a given sentence $w_1, w_2, \ldots w_n$ of length $n$. This is because any rules which contain a lexical item that is not one of $w_1 \ldots w_n$, can be safely discarded.

- The result: we can parse in $O(n^5|N|^3)$ time.
Accurate Unlexicalized Parsing
Accurate Unlexicalized Parsing

PCFG Problem 2
Lack of sensitivity to structural information
Accurate Unlexicalized Parsing

PCFG Problem 2
Lack of sensitivity to structural information

Solution

- This problem is also solved by lexicalization.
  - (maybe that’s the main problem that’s being solved by lexicalization)
PCFG Problem 2
Lack of sensitivity to structural information

Solution

- This problem is also solved by lexicalization.
  - (maybe that’s the main problem that’s being solved by lexicalization)
- But can we do without lexicalizing the grammar?
5. Accurate Unlexicalized Parsing: PCFGs and Independence

- The symbols in a PCFG define independence assumptions:

  \[ S \rightarrow NP \, VP \]

  \[ NP \rightarrow DT \, NN \]

- At any node, the material inside that node is independent of the material outside that node, given the label of that node.

- Any information that statistically connects behavior inside and outside a node must flow through that node.
Michael Collins (2003, COLT)

Independence Assumptions

- PCFGs

```
s
  np
    dt
      the
    n
      lawyer
  vp
    v
      questioned
    np
      dt
        the
      n
        witness
```

- Lexicalized PCFGs

```
s(questioned)
  np(lawyer)
    dt
      the
    n
      lawyer
  vp(questioned)
    v
      questioned
    np(witness)
      dt
        the
      n
        witness
```
Non-Independence I

- Independence assumptions are often too strong.

- Example: the expansion of an NP is highly dependent on the parent of the NP (i.e., subjects vs. objects).
We can relax independence assumptions by encoding dependencies into the PCFG symbols:

- What are the most useful features to encode?

Parent annotation [Johnson 98]

Marking possessive NPs

- What are the most useful features to encode?
A Fully Annotated Tree

```
ROOT
  \[ S^\text{ROOT-v} \]
  \[
  "^S \quad NP^S-B \quad VP^S-VBF-v \quad .^S "^S
  \]
  \[
  \quad DT-U^NP \quad VBZ^BE^VP \quad NP^VP-B \quad ! "
  \]
  \[
  \quad This \quad is \quad NN^NP \quad NN^NP \quad panic \quad buying
  \]
```
Final Test Set Results

<table>
<thead>
<tr>
<th>Parser</th>
<th>LP</th>
<th>LR</th>
<th>F1</th>
<th>CB</th>
<th>0 CB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magerman 95</td>
<td>84.9</td>
<td>84.6</td>
<td>84.7</td>
<td>1.26</td>
<td>56.6</td>
</tr>
<tr>
<td>Collins 96</td>
<td>86.3</td>
<td>85.8</td>
<td>86.0</td>
<td>1.14</td>
<td>59.9</td>
</tr>
<tr>
<td>Klein &amp; M 03</td>
<td>86.9</td>
<td>85.7</td>
<td>86.3</td>
<td>1.10</td>
<td>60.3</td>
</tr>
<tr>
<td>Charniak 97</td>
<td>87.4</td>
<td>87.5</td>
<td>87.4</td>
<td>1.00</td>
<td>62.1</td>
</tr>
<tr>
<td>Collins 99</td>
<td>88.7</td>
<td>88.6</td>
<td>88.6</td>
<td>0.90</td>
<td>67.1</td>
</tr>
</tbody>
</table>

- Beats “first generation” lexicalized parsers.
Automatic Annotation
(latent variable grammars)

[Matsuzaki et al. ’05, Petrov et al. ’06]
Constituency Parsing

- Decompose tree into parts (grammar rules)
- Assign score to each rule
- Find best scoring tree using CKY
- Maybe do some tricks for better scoring of rules
Dependency Parsing
Dependency Representation
Head Words

Each constituent has one word which captures its “essence”.

- (S John **saw** the young boy with the large hat)
- (VP **saw** the young boy with the large hat)
- (NP the young **boy** with the large hat)
- (NP the large **hat**)
- (PP **with** the large hat)
  - **hat** is the “semantic head”
  - **with** is the “functional head”
  - (it is common to choose the functional head)
Adding Headwords to Trees

S

NP

DT (the)  NN (lawyer)

VP

Vt (questioned)

NP

DT (the)  NN (witness)

↓

S (questioned)

NP (lawyer)

DT (the)  NN (lawyer)

VP (questioned)

Vt (questioned)

NP (witness)

DT (the)  NN (witness)
If we take the head-annotated trees and “forget” about the constituents, we get a representation called “dependency structure”.

Dependency structure capture the relation between words in a sentence.
Dependency representation is very common. We will return to it in the future.
Dependency Representation

Dependency representation is very common. We will return to it in the future.
Dependency representation is very common. We will return to it in the future.
Dependency Representation

questioned

lawyer  witness

the  the
Dependency Representation

The lawyer questioned the witness.

The lawyer questioned the witness.
I heard that the lawyer questioned the witness under oath yesterday.
Dependency Representation

I heard that the lawyer questioned the witness under oath yesterday
Dependency Representation

Projectivity

**Projective Tree** (no crossing arcs):

I heard that the lawyer questioned the witness under oath yesterday

**Non-Projective Tree** (crossing arcs):

root → John → saw → a → dog → yesterday → which → was → a → Yorkshire Terrier
Dependency Representations

There are many different dependency representations

- Different choice of heads.
- Different set of labels.
- Each language usually has its own treebank, with own choices
Universal Dependencies

- A multi-national project aiming at producing a consistent set of dependency annotations in many (all!) languages.

Why is this good? why is this interesting?

Interesting project/research idea: are the annotations really consistent across languages? do languages differ only in word order?
Universal Dependencies

- A multi-national project aiming at producing a consistent set of dependency annotations in many (all!) languages.
- Abstract over linguistic differences.
- Same set of parts-of-speech and morphology features.
- Same dependency relations.
- Same choice of heads.
Three main approaches to Dependency Parsing

Conversion

- Parse to constituency structure.
- Extract dependencies from the trees.

Global Optimization (Graph based)

- **Define** a scoring function over <sentence, tree> pairs.
- **Search** for best-scoring structure.
- Simpler scoring ⇒ easier search.
- (Similar to how we do tagging, constituency parsing.)

Greedy decoding (Transition based)

- Start with an unparsed sentence.
- Apply **locally-optimal** actions until sentence is parsed.
Graph-based Parsing
The Parsing Problem

\[ \text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \text{score}(y, x) \]

search over all possible parses
and find the one with the highest score
The Parsing Problem

\[
\text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \text{score}(y, x)
\]

search over all possible parses and find the one with the highest score

Training objective

\[
\text{score}(y, x) > \text{score}(y', x) \quad \forall y \neq y'
\]
The Parsing Problem

\[ \text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \text{score}(y, x) \]

search over all possible parses and find the one with the highest score

**Challenge**: very hard search problem
The Parsing Problem

\[ \text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \text{score}(y, x) \]

search over all possible parses and find the one with the highest score

**Challenge:** very hard search problem

**Solution:** decompose score:

\[ \text{score}(y, x) = \sum_{p \in y} \text{score}(p) \]
Structured Prediction Recipe

\[
predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p))
\]

- Decompose structure to local factors.
- Assign a score to each factor.
- Structure score = sum of local scores.
- Look for highest scoring structure.
Structured Prediction Recipe

\[ \text{predict}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p)) \]
First-order parser ("MST")

- **Goal**: predict a parse tree.

- **Parts**: \(<\text{head}, \text{modifier}>\) pairs (arcs).

\[
\text{parse}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{(h,m) \in y} \text{score}(\phi(h, m, x))
\]
Structured Prediction Recipe

\[ \text{predict}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p)) \]
Decomposing

(score( the/D fox/N who/P likes/V apples/N jumped/V over/P a/D dog/N ) =

=score(fox, the)
+score(fox, who)
+score(who, likes)
+score(likes, apples)
+score(jumped, fox)
+score(jumped, over)
+score(over, dog)
+score(dog, a)
Input Sentence: ”They ate pizza”

score each possible arc (n²)
Graph-based Parsing (Inference)

Spanning tree with maximal score

Eliyahu Kiperwasser (Bar-Ilan University)
Structured Prediction Recipe

\[ \text{predict}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p)) \]
Finding the Best Tree

• Projective Parsing:
  • **The Eisner Algorithm** (Dynamic Programming)

• Non-projective parsing:
  • **Directed Spanning Tree**
    (Chu Liu Edmunds, Tarjan)
Eisner Parsing Algorithm
Eisner Parsing Algorithm

- Finding the best **projective** tree.

- We can use the lexicalized version of CKY
  
  - ... but this will give use \( n^5 \) algorithm.

- Jason Eisner (and Giorgio Satta) reduced this to \( n^3 \) using a more specialized algorithm.

  - Main idea: decouple scoring of left and right modifiers.
Dependency Parsing Algorithm - First-order Model

(h, m) ← (h, r) + (r + 1, m)

(h, e) ← (h, m) + (m, e)

(slide from Alexander Rush)
Base Case

* As McGwire neared, fans went wild

(slide from Alexander Rush)
As McGwire neared, fans went wild.
As McGwire neared, fans went wild.

(slide from Alexander Rush)
* As McGwire neared, fans went wild.
As McGwire neared, fans went wild.
* As McGwire neared, fans went wild.

(slide from Alexander Rush)
As McGwire neared, fans went wild.
* As McGwire neared, fans went wild.

(slide from Alexander Rush)
As McGwire neared, fans went wild.
Eisner's algorithm

- Start with two triangles for each word.
- Combine two small triangles into a larger trapezoid, and add arc between the words.
- Combine trapezoid and triangle into larger triangle.
- Need an order that computes all smaller parts before larger ones.
Algorithm Key

- L; left-facing item
- R; right-facing item
- C; completed item (triangle)
- I; incomplete item (trapezoid)
Algorithm

Initialize:
for $i$ in $0 \ldots n$ do
  $\pi[C, L, i, i] = 0$
  $\pi[C, R, i, i] = 0$
  $\pi[I, L, i, i] = 0$
  $\pi[I, R, i, i] = 0$

Inner Loop:
for $k$ in $1 \ldots n$ do
  for $s$ in $0 \ldots n$ do
    $t \leftarrow k + s$
    if $t \geq n$ then break
    $\pi[I, L, s, t] = \max_{r \in s \ldots t-1} \pi[C, R, s, r] + \pi[C, L, r + 1, t]$
    $\pi[I, R, s, t] = \max_{r \in s \ldots t-1} \pi[C, R, s, r] + \pi[C, L, r + 1, t]$
    $\pi[C, L, s, t] = \max_{r \in s \ldots t-1} \pi[C, L, s, r] + \pi[I, L, r, t]$
    $\pi[C, R, s, t] = \max_{r \in s+1 \ldots t} \pi[I, R, s, r] + \pi[C, R, r, t]$
  return $\pi[C, R, 0, n]$
Structured Prediction Recipe

\[
predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p))
\]
Representing

\[
predict(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p))
\]

- *feature function* extracts useful signals from parts.
- most work traditionally goes into this component.
Linear features

\[ \phi(\text{saw}, \text{with}) \]

- Identities of the words \( w_i \) and \( w_j \) and the label \( l_k \)
- Part-of-speech tags of the words \( w_i \) and \( w_j \) and the label \( l_k \)
- Part-of-speech of words surrounding and between \( w_i \) and \( w_j \)
- Number of words between \( w_i \) and \( w_j \), and their orientation
- Combinations of the above
Linear features

\begin{verbatim}
[went]
[VERB]
[went, As]
[VERB, IN]
[ADJ, *, ADP]
[NNS, VBD, ADP]
[NNS, ADP, NNP]
[ADP, left, 5]
[JJ, *, IN]
[NOUN, VERB, IN]
[NOUN, IN, NOUN]
[IN, left, 5]
[NNS, VBD, ADP, NNP]
[went, VERB, As, IN]
[went, VERB, left, 5]
[went, As, ADP, left, 5]
[VBD, ADP, ADP, left, 5]
[ADJ, ADP, NNP, left, 5]
[VERB, As, IN, left, 5]
[VERB, *, IN, left, 5]
[VBD, As, ADP]
[VBD, *, ADP]
[NNS, VBD, *]
[NNS, VBD, NNP]
[VERB, As, IN]
[VERB, *, IN]
[NOUN, VERB, *]
[NOUN, VERB, NOUN]
[went, VBD, As, ADP]
[went, VBD, left, 5]
[VERB, JJ, *, IN]
[VBD, *, ADP]
[VBD, As, ADP]
[VBD, ADP, ADP, left, 5]
[VBD, ADP, NNP, left, 5]
[went, As, IN, left, 5]
[VERB, JJ, IN, left, 5]
[VBD, ADJ, *, ADP]
[VBD, ADP, NNP, left, 5]
[went, VERB, IN, left, 5]
[VERB, JJ, *, IN]
[VERB, NOUN, VERB, *]
[VERB, NOUN, NOUN]
[went, VERB, As, ADP]
[went, As, ADP, left, 5]
[went, As, left, 5]
[VBD, ADP, left, 5]
[VBD, ADP, NNP, left, 5]
[VERB, As, IN, left, 5]
[VERB, *, IN, left, 5]
[As]
[IN]
[went, VERB]
[went, As, ADP]
[ADP]
[went, VBD]
[As, IN]
[As, ADP]
[went, VBD, As, IN]
[went, VBD, As, left, 5]
[went, VBD, ADP, left, 5]
[went, VBD, ADP, NNP, left, 5]
[went, VBD, As, IN, left, 5]
[went, VBD, JJ, IN, left, 5]
[went, VBD, ADJ, As, left, 5]
[went, VBD, ADP, As, left, 5]
[went, VBD, ADP, NNP, As, left, 5]
[went, VBD, ADP, NNP, *]
[went, VBD, ADP, NNP, NNP]
[went, VBD, ADP, NNP, left, 5]
[went, VBD, ADP, NNP, *]
[went, VBD, ADP, NNP, NNP]
[went, VBD, ADP, NNP, left, 5]
[went, VBD, ADP, NNP, *]
[went, VBD, ADP, NNP, NNP]
[went, VBD, ADP, NNP, left, 5]
[went, VBD, ADP, NNP, *]
[went, VBD, ADP, NNP, NNP]
[went, VBD, ADP, NNP, left, 5]

\end{verbatim}

(slide from Slav Petrov)
Neural Features (bi-LSTM)

\( \phi(x, jumped, fox) \)

the \( /D \quad \) fox \( /N \quad \) who \( /P \quad \) likes \( /V \quad \) apples \( /N \quad \) jumped \( /V \quad \) over \( /P \quad \) a \( /D \quad \) dog \( /N \quad \) fox \( /N \quad \) jumped

\[ \phi(x, h, m) = [BIRNN(x, h); BIRNN(x, m)] \]
Structured Prediction Recipe

\[ \text{predict}(x) = \arg\max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p)) \]

- search
- decompose
- represent
- score
Scoring

\[\text{predict}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p))\]

- a model (linear or not) assigns a score based on features

\[
\text{score}(\phi(h, m)) = w \cdot \phi(h, m)
\]
Scoring

\[ \text{predict}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p)) \]

- a model (linear or not) assigns a score based on features

**Linear:** \[ \text{score}(\phi(h, m)) = \mathbf{w} \cdot \phi(h, m) \]

**Non-linear:** \[ \text{score}(\phi(h, m)) = \mathbf{w} \cdot \tanh(\mathbf{U} \cdot \phi(h, m)) \]
Scoring

\[
\text{predict}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{p \in y} \text{score}(\phi(p))
\]

- a model (linear or not) assigns a score based on features

**Linear:**  
\[
\text{score}(\phi(h, m)) = w \cdot \phi(h, m)
\]

**Non-linear:**  
\[
\text{score}(\phi(h, m)) = w \cdot \tanh(U \cdot \phi(h, m))
\]

**Bi-linear:**  
\[
\text{score}(\phi(h, m)) = \phi(h)W\phi(m)
\]
Training
(linear, structured perceptron)

• For each gold pair \((x, y)\):

  • Predict \(\hat{y}\) based on \(x\).

  • if \(\hat{y} \neq y\), update:

\[
\begin{align*}
\mathbf{w} & \leftarrow \mathbf{w} + \sum_{h,m \in y} \phi(h,m) - \sum_{h,m \in \hat{y}} \phi(h,m)
\end{align*}
\]

\[
\text{predict}(x) = \arg \max_{y \in \mathcal{Y}(x)} \sum_{h,m \in y} \mathbf{w} \cdot \phi(h,m)
\]
Training
(linear, structured perceptron)

• For each gold pair \((x, y)\):
  
  • Predict \(\hat{y}\) based on \(x\).
  
  • if \(\hat{y} \neq y\), update:

\[
predict(x) = \arg\max_{y \in \mathcal{Y}(x)} \sum_{h,m \in y} w \cdot \phi(h,m)
\]

\[
w \leftarrow w + \sum_{h,m \in y} \phi(h,m) - \sum_{h,m \in \hat{y}} \phi(h,m)
\]

gold features \hspace{1cm} predicted features
At Parsing Time

- Assign a score to each head, modifier pair.
- Use Eisner/CLE to find best scoring tree.
Transition-based parsing
Transition-based (greedy) parsing

1. Start with an unparsed sentence.

2. Apply **locally-optimal** actions until sentence is parsed.
Transition-based (greedy) parsing

1. Start with an unparsed sentence.
2. Apply locally-optimal actions until sentence is parsed.
3. Use whatever features you want.
4. Surprisingly accurate.
5. Can be extremely fast.
An abstract machine composed of a stack and a buffer.

Machine is initialized with the words of a sentence.

A set of actions process the words by moving them from buffer to stack, removing them from the stack, or adding links between them.

A specific set of actions define a transition system.
The Arc-Eager Transition System

- **SHIFT** move first word from buffer to stack.
  (pre: Buffer not empty.)

```
A
B
C
D
```

```
A
B
C
D
```
The Arc-Eager Transition System

- **SHIFT** move first word from buffer to stack.
  
  (pre: Buffer not empty.)

- **LEFTARC** label make first word in buffer head of top of stack, pop the stack.
  
  (pre: Stack not empty. Top of stack does not have a parent.)
The Arc-Eager Transition System

- **SHIFT** move first word from buffer to stack.
  (pre: Buffer not empty.)

- **LEFTARC_{label}** make first word in buffer head of top of stack, pop the stack.
  (pre: Stack not empty. Top of stack does not have a parent.)

- **RIGHTARC_{label}** make top of stack head of first in buffer, move first in buffer to stack.
  (pre: Buffer not empty.)
The Arc-Eager Transition System

- **SHIFT** move first word from buffer to stack.
  (pre: Buffer not empty.)

- **LEFTARC** make first word in buffer head of top of stack, pop the stack.
  (pre: Stack not empty. Top of stack does not have a parent.)

- **RIGHTARC** make top of stack head of first in buffer, move first in buffer to stack.
  (pre: Buffer not empty.)

- **REDUCE** pop the stack
  (pre: Stack not empty. Top of stack has a parent.)
She ate pizza with pleasure.
She ate pizza with pleasure
She ate pizza with pleasure.
She ate pizza with pleasure.
She ate pizza with pleasure.
Parsing Example

She ate pizza with pleasure
She ate pizza with pleasure.
She ate pizza with pleasure.
She ate pizza with pleasure
She ate pizza with pleasure.
She ate pizza with pleasure
What do we know about the arc-eager transition system?

- Every sequence of actions result in a valid projective structure.
- Every projective tree is derivable by (at least one) sequence of actions.
- Given a tree, finding a sequence of actions for deriving it. ("oracle")

we know these things also for the arc-standard, arc-hybrid and other transition systems
This knowledge is quite powerful

Parsing with an oracle sequence

```java
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
```
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
She LEFT She RIGHT RE RIGHT RIGHT RE RE RE
```

```
She ate pizza with pleasure
```
This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
SH  LEFT  SH  RIGHT  RE  RIGHT  RIGHT  RIGHT  RE  RE  RE
```

She ate pizza with pleasure
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”
This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
  action ← sequence.next()
  configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
This knowledge is quite powerful

Parsing with an oracle sequence

\[
\text{sequence} \leftarrow \text{oracle}(\text{sentence}, \text{tree}) \\
\text{configuration} \leftarrow \text{initialize}(\text{sentence}) \\
\text{while} \not\text{configuration.IsFinal()} \text{ do} \\
\quad \text{action} \leftarrow \text{sequence.next()} \\
\quad \text{configuration} \leftarrow \text{configuration.apply}(\text{action}) \\
\text{return} \ \text{configuration.tree}
\]

“She ate pizza with pleasure”

\[\text{SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE} \]

\[\text{She ate pizza with pleasure}\]
This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
  action ← sequence.next()
  configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

Sh Left Sh Right Re Right Right Re Re Re Re
This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

She Left Sh Right Re Right Right Right Re Re Re Re
This knowledge is quite powerful

Parsing with an oracle sequence

```plaintext
sequence ← oracle(sentence, tree)  
configuration ← initialize(sentence) 
while not configuration.IsFinal() do  
    action ← sequence.next()  
    configuration ← configuration.apply(action) 
return configuration.tree
```

“She ate pizza with pleasure”

```
SH    LEFT    SH    RIGHT    RE    RIGHT    RIGHT    RIGHT    RE    RE    RE
```

She ate pizza with pleasure
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
SH  LEFT  SH  RIGHT  RE  RIGHT  RIGHT  RE  RE  RE
```

She ate pizza with pleasure
This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”

SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
This knowledge is quite powerful

Parsing with an oracle sequence

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree

“She ate pizza with pleasure”
This knowledge is quite powerful

Parsing with an oracle sequence

```python
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
Sh Left Sh Right Re Right Right Right Re Re Re
```

```
She    ate   pizza   with   pleasure
```
This knowledge is quite powerful

 Parsing with an oracle sequence

\[
\text{sequence} \leftarrow \text{oracle}(\text{sentence}, \text{tree}) \\
\text{configuration} \leftarrow \text{initialize}(\text{sentence}) \\
\text{while not configuration.IsFinal()} \text{ do} \\
\quad \text{action} \leftarrow \text{sequence.next()} \\
\quad \text{configuration} \leftarrow \text{configuration.apply}(\text{action}) \\
\text{return configuration.tree}
\]

“She ate pizza with pleasure”

She Left Sh Right Re Right Right Right Re Re Re Re

She ate pizza with pleasure
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
```

She ate pizza with pleasure
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
She  ate  pizza  with  pleasure
```
This knowledge is quite powerful

Parsing with an oracle sequence

\[
\text{sequence} \leftarrow \text{oracle}(\text{sentence}, \text{tree}) \\
\text{configuration} \leftarrow \text{initialize}(\text{sentence}) \\
\textbf{while} \not\text{configuration.\text{IsFinal}()} \textbf{do} \\
\hspace{1em} \text{action} \leftarrow \text{sequence.\text{next}()} \\
\hspace{2em} \text{configuration} \leftarrow \text{configuration.\text{apply}(\text{action})} \\
\textbf{return} \text{configuration.\text{tree}}
\]

“She ate pizza with pleasure”

```
SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
```

She ate pizza with pleasure
This knowledge is quite powerful

Parsing with an oracle sequence

```python
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
```

She ate pizza with pleasure
This knowledge is quite powerful

Parsing with an oracle sequence

```python
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
```

She ate pizza with pleasure
This knowledge is quite powerful

Parsing with an oracle sequence

```
sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()
    configuration ← configuration.apply(action)
return configuration.tree
```

“She ate pizza with pleasure”

```
SH LEFT SH RIGHT RE RIGHT RIGHT RE RE RE
```

She ate pizza with pleasure
This knowledge is quite powerful

Parsing without an oracle

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
  action ← sequence.next()
  configuration ← configuration.apply(action)
return configuration.tree
This knowledge is quite powerful

Parsing without an oracle

\textbf{start with weight vector} \( w \)
\textbf{configuration} \( \leftarrow \) \textbf{initialize(sentence)}
\textbf{while} not \textbf{configuration}.\textbf{IsFinal()} \textbf{do}
  \textbf{action} \( \leftarrow \) \textbf{predict}(w, \phi(\textbf{configuration}))
  \textbf{configuration} \( \leftarrow \) \textbf{configuration}.\textbf{apply}(\textbf{action})
\textbf{return} \textbf{configuration}.\textbf{tree}
This knowledge is quite powerful

Parsing without an oracle

summarize the configuration as a feature vector

start with weight vector $w$
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← predict($w$, $\phi$(configuration))
    configuration ← configuration.apply(action)
return configuration.tree
This knowledge is quite powerful

Parsing without an oracle

summarize the configuration as a feature vector

start with weight vector $w$
configuration $\leftarrow$ initialize(sentence)
while not configuration.IsFinal() do
    action $\leftarrow$ predict($w$, $\phi$(configuration))
configuration $\leftarrow$ configuration.apply(action)
return configuration.tree

predict the action based on the features
This knowledge is quite powerful

Parsing without an oracle

- summarize the configuration as a feature vector
- start with weight vector $w$
- configuration $\leftarrow$ initialize(sentence)
- while not configuration.IsFinal() do
  - action $\leftarrow$ predict($w$, $\phi$(configuration))
  - configuration $\leftarrow$ configuration.apply(action)
- return configuration.tree

predict the action based on the features

need to learn the correct weights
This knowledge is quite powerful

Parsing with an oracle sequence

sequence $\leftarrow$ oracle(sentence, tree)
configuration $\leftarrow$ initialize(sentence)
while not configuration.IsFinal() do
    action $\leftarrow$ sequence.next()
    configuration $\leftarrow$ configuration.apply(action)
This knowledge is quite powerful

Learning a parser

sequence ← oracle(sentence, tree)
configuration ← initialize(sentence)
while not configuration.IsFinal() do
    action ← sequence.next()

configuration ← configuration.apply(action)
This knowledge is quite powerful

Learning a parser

\[ w \leftarrow 0 \]

for sentence, tree pair in corpus do

sequence \leftarrow \text{oracle}(\text{sentence, tree})

configuration \leftarrow \text{initialize}(\text{sentence})

while not configuration.IsFinal() do

action \leftarrow \text{sequence}.next()

features \leftarrow \phi(\text{configuration})

predicted \leftarrow \text{predict}(w, \phi(\text{configuration}))

if predicted \neq action then

w.update(\phi(\text{configuration}), action, predicted)

configuration \leftarrow \text{configuration}.apply(action)

return w
This knowledge is quite powerful

Parsing time

configuration ← initialize(sentence)
while not configuration.isFinal() do
    action ← predict(w, ϕ(configuration))
    configuration ← configuration.apply(action)
return configuration.tree
In short

- Summarize configuration by a set of features.
- Learn the best action to take at each configuration.
- Hope this generalizes well.
Transition Based Parsing

- A different approach.
- Very common.
- Can be as accurate as first-order graph-based parsing.
  - Higher-order graph-based are still better.
- Easy to implement.
- Very fast. \((O(n))\)
- Can be improved further:
  - Easy-first
  - Dynamic oracle
  - Beam Search
Summary

- Syntax (hierarchical structure)
- Grammars, PCFG, CKY Algorithm
  - Head Words
- Constituency to dependency with head-words
- Dependency Parsing
  - Graph Based
  - Transition Based
Parsers

- Phrase Based
  - Berkeley Parser
  - Stanford Parser

- Dependency
  - spaCy
  - Stanford Parser

- Dependency + research
  - BIST parser (Kiperwasser and Goldberg)
  - Stack LSTM parser (Dyer et al)