Ambiguity is massive

- For parsing related problems, we have to keep in mind that ambiguity is massive:

- The number of parses grows indeed exponentially with the length of the sentence.
For sequence prediction, dynamic programming relies on a computation DAG.

For parsing, dynamic programming relies on an oriented hypergraph (DAH).

Basic representations:

1. Sentence indexation:
   - Example: \( \langle \text{NP}, 0, 2 \rangle \) means that we recognized an NP between positions 0 and 2.

2. A span is a triple \( \langle X, i, j \rangle \) that records the fact that a parser already found at least one parse with root \( X \) covering indexes \( i \) to \( j \):
   - Example: \( \langle \text{NP}, 0, 2 \rangle \) means that we recognized an NP between positions 0 and 2.

3. Spans define equivalence classes between subparses.
In general a span $\langle X, i, j \rangle$ is the result of a deduction from a sequence of spans with an inference of the form:

$$
\frac{\langle Y_1, i, k_1 \rangle \ldots \langle Y_m, k_{m-1}, j \rangle}{\langle X, i, j \rangle} \quad X \rightarrow Y_1 \ldots Y_m \in R
$$

The spans can be seen as the nodes $V$ of an hypergraph whose hyperedges $E$ encode instanciated grammar rules of the form:

$$
\langle X, i, j \rangle, \langle Y_1, i, k_1 \rangle \ldots \langle Y_m, i, k_{m-1} \rangle
$$

- head
- tail
Example

\[ V = \{ \langle 0, 4, X \rangle, \langle 0, 3, X \rangle, \langle 0, 2, X \rangle, \langle 1, 3, X \rangle, \langle 0, 1, w_1 \rangle, \langle 1, 2, w_2 \rangle, \langle 2, 3, w_3 \rangle, \langle 3, 4, w_4 \rangle \} \]

\[ E = \{ (\langle 0, 4, X \rangle, \langle 0, 3, X \rangle \langle 3, 4, w_4 \rangle), (\langle 0, 3, X \rangle, \langle 0, 2, X \rangle \langle 2, 3, w_3 \rangle), (\langle 0, 3, X \rangle, \langle 0, 1, w_1 \rangle \langle 1, 3, X \rangle), (\langle 0, 2, X \rangle, \langle 0, 1, w_1 \rangle \langle 1, 2, w_2 \rangle), (\langle 1, 3, X \rangle, \langle 1, 2, w_2 \rangle \langle 2, 3, w_3 \rangle), (\langle 0, 1, w_1 \rangle, \epsilon), (\langle 1, 2, w_2 \rangle, \epsilon), (\langle 2, 3, w_3 \rangle, \epsilon), (\langle 3, 4, w_4 \rangle, \epsilon) \} \]
Shared forest

- The **shared forest** or **parse forest** is the data structure encoding the set of trees generating the sentence $w_0 \ldots w_n$ with grammar $G$ as a directed acyclic hypergraph.
- This data structure is also the result of the **intersection theorem** of (Bar Hillel 1964): the intersection of a context free language with a regular language is a context free language. The proof of the theorem is constructive and yields an intersection grammar that we express by the hypergraph.
- The shared forest has the same status as the dynamic programming **DAG** used in sequence modelling (see class 2).

Relation to parsing algorithms

Parsing algorithms can be seen as procedures building such a parse forest (most of the times partially). They essentially differ by the strategy (such as top down, bottom up) with which they achieve the result.
Some definitions

Definition (Hypergraph)
An hypergraph $H = \langle V, E \rangle$ is a couple where $V$ is a set of nodes and $E$ a set of hyperarcs. An hyperarc $e \in E$ is a couple $e = \langle h, v_1 \ldots v_k \rangle$ where $h$ is the head and $v_1 \ldots v_k \in V^k$ is the queue.

- We note $\text{head}(e)$ the function returning the head of an hyperedge and $\text{tail}(e)$ the function returning its tail.
- We note $|e| = k$ the arity of an hyperedge.

Definition (Incoming Edges)
Let $v \in V$ be a node of an hypergraph $H$, the set $IE(v) = \{ e \in E | v = \text{head}(e) \}$ is the set of hyperarcs whose head is $v$.

Definition (Outgoing Edges)
Let $v \in V$ be a node of an hypergraph $H$, the set $OE(v) = \{ e \in E | v \in \text{tail}(e) \}$ is the set of hyperarcs whose tail contains $v$. 
Derivations

Definition (Derivation)

A derivation $D(v)$ rooted at $v$ is defined recursively as follows:

- if $e \in IE(v)$ and $|e| = 0$ then $D(v) = \langle v, \epsilon \rangle$ is a derivation
- if $e \in IE(v)$ and $|e| > 0$ then $D(v) = \langle v, D(v_1) \ldots D(v_k) \rangle$

A derivation is the counterpart of a path in a DAG (as used in class 2). It also encodes a parse tree
Weighted hypergraphs

Definition (Derivation weight)

For weighted hypergraphs, we use an edge scoring function $\psi : E \mapsto \mathbb{R}$. The weight $\sigma(D(v))$ of a derivation rooted in $v$ is defined recursively as:

- **Base** if $e \in IE(v)$ and $|e| = 0$ then $\sigma(D(v)) = 1$
- **Recurrence** if $e \in IE(v)$ and $|e| > 0$ then:

\[
\sigma(D(v)) = \psi(e) \times \prod_{i=1}^{k} \sigma(D(v_i))
\]

Definition (Maximum weight from source)

The maximal weight from a source $\delta(v)$ is the maximal weight of all derivations $D(v) \in \mathcal{D}(v)$ rooted in $v$:

\[
\delta(v) = \begin{cases} 
1 & \text{if } v \text{ has no incoming edge} \\
\max_{D(v) \in \mathcal{D}(v)} \sigma(D(v)) & \text{otherwise}
\end{cases}
\]
Plan

1. Context free grammar parsing algorithms
2. Transition based parsing for dependency grammar
3. Dynamic programming for dependency grammar
4. CKY for tree adjoining grammars
**Binary trees**

- **CKY** algorithm is usually framed as an algorithm for grammars with rules of arity 2.

- In general, the grammar is induced from a treebank. We can turn a treebank with trees of arbitrary arity into a treebank in **binary form** by using two transforms:

  - **rule binarization**:

  ```
  X  A_1 A_2 A_3 A_4  
   |   a    a    a    a |

  X  A_1 A_2 + A_3 + A_4  
   |   a            a |
   |   A_2 A_3 + A_4 |
   |       a         |

  X  A_1 + A_2 + A_3 A_4  
   |       a       |
   |   A_1 A_2     |
   |       a   a   |

  Original tree | Right binarization | Left binarization
  --------------|-------------------|------------------
  
  ```

  - **unary reduction**: for each symbol $A$ on the top of an unary production, replace it by the sequence of symbols $A \rightarrow \ldots \rightarrow X$ leading (a) to the next binary production or (b) to a leaf.
The Cocke Kasami Younger algorithm

We suppose a sentence $w_0 \ldots w_n$ to parse and a grammar $G$ in binary form.

The **CKY** algorithm is a popular algorithm for building a shared forest, it consists of two inference rules:

- **scan**:
  \[
  \langle w_i, i, i+1 \rangle \quad (0 \leq i \leq n)
  \]

- **reduce**:
  \[
  \frac{\langle Y_1, i, k \rangle \langle Y_2, k, j \rangle}{\langle X, i, j \rangle} \quad X \rightarrow Y_1 Y_2 \in R
  \]

- and a **goal**:
  \[
  \langle X, 0, n+1 \rangle
  \]

**ordering strategy** the algorithm proceeds by building nodes (parse items) of the shared forest following a topological order on the DAH. Typically it amounts to build items with spans of increasing length follows a valid topological ordering.
Definition (Projected graph)

The projected graph of an hypergraph $H = \langle V, E \rangle$ is the directed graph $G = \langle V, E' \rangle$ such that $E' = \{(u, v) | \exists e \in IE(v) \land u \in \text{tail}(v)\}$. If $H$ is acyclic then $G$ is acyclic.

Example:

Definition (Topological order on an hypergraph)

A topological order for $H$ is a total ordering of its nodes which is a topological order for its projected graph $G$. 
Weighted inference rules

- We suppose a sentence $w_0 \ldots w_n$ to parse and a grammar $G$ in binary form.

- The **CKY** algorithm is a popular algorithm for building a shared forest, it consists of two inference rules:
  - **scan**:
    \[
    \langle w_i, i, i+1, 1.0 \rangle \\
    (0 \leq i \leq n)
    \]
  - **reduce**:
    \[
    \frac{\langle Y_1, i, k, \delta_1 \rangle \langle Y_2, k, j, \delta_2 \rangle}{\langle X, i, j, \psi(X, Y_1, Y_2) \times \delta_1 \times \delta_2 \rangle} \\
    X \rightarrow Y_1 Y_2 \in R
    \]
  - and a **goal**:
    \[
    \langle X, 0, n+1, \delta \rangle
    \]
Viterbi-CKY algorithm

\begin{function}
\textsc{Viterbi-CKY}(w_0 \ldots w_n)
T \leftarrow \emptyset
\text{for } 0 \leq i \leq n \text{ do }
\quad \delta(\langle w_i, i, i+1 \rangle) \leftarrow 1.0
\text{end for}
\text{for all } \langle X, i, j \rangle \in V \text{ following a topological order do }
\quad \delta(\langle X, i, j \rangle) \leftarrow 0
\quad \text{for } \langle X, i, j \rangle \rightarrow \langle Y_1, i, k \rangle \langle Y_2, k, j \rangle \in IE(v) \text{ do }
\quad \quad \delta(\langle X, i, j \rangle) \leftarrow \max (\delta(v), \psi(e) \times \delta(\langle Y_1, i, k \rangle) \times \delta(\langle Y_2, k, j \rangle))
\quad \text{end for}
\text{end for}
\text{end function}
\end{function}

Comments
Note that the topological order is usually implemented by computing iteratively all spans of length $t + 1$ once all spans of length $t$ have already been computed.
The core loop involves iteratively performing the reduction inference:

where \( i, j, k \) all have range \( n \). Thus we have a time complexity in \( \mathcal{O}(n^3) \)
Example
finding the probability of the best parse with CRF

- Let $y$ be a derivation $D(v)$, a CRF for binary trees takes the following form:

$$P(y|x) = \frac{\exp(w\Phi(x, y))}{\sum_{y'} \exp(w\Phi(x, y'))}$$

- Which can be decomposed as:

$$P(y|x) = \prod_{e \in D(v)} \psi(e, x)$$

- Or more succinctly:

$$P(y|x) = \frac{\prod_{e \in D(v)} \psi(e, x)}{\sum_{D(v')} \prod_{e' \in D(v')} \psi(e', x)}$$

- CRF is a statistical model whose weights are positive reals, that is

$$\psi : E \rightarrow \mathbb{R}^+$$
Finding the probability of the best parse

- For **CRF** solving the argmax, amounts to find:

  \[ \hat{y} = \text{argmax}_{D(v')} \prod_{e' \in D(v')} \psi(e', x) \]

- Computing the normalisation constant, requires to evaluate:

  \[ Z = \sum_{D(v')} \prod_{e' \in D(v')} \psi(e', x) \]

- One can run Viterbi-CKY to get the result, example:

  The result is 30 for argmax and 40 for the normalisation value. For the latter, the algebra is different: we have to use sum instead of max in Viterbi-CKY. The resulting probability is \( p = \frac{30}{40} \)
Remark on features and scoring functions

- We saw (class 3) that PCFG is insufficiently lexicalized.
- Many machine learning methods vectorize the parse edges of the form:

\[ e : \langle X, i, j \rangle \rightarrow \langle Y_1, i, k \rangle \langle Y_2, k, j \rangle \]

this means that the scoring function \( \psi(e, x) \) has not only access to the categories \( X, Y_1, Y_2 \) of the edge but also to lexical words \( x \) by means of the indexed positions \( i, j, k \).

```
X
  / \  /
Y_1 Y_2
  \   \
    /  /
  w_i ... w_k ... w_j
  ```
Some results

<table>
<thead>
<tr>
<th>NAME</th>
<th>F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCFG – base</td>
<td>~ 70</td>
</tr>
<tr>
<td>Unlex. annotations</td>
<td>85.8</td>
</tr>
<tr>
<td>PCFG-LA</td>
<td>90.1</td>
</tr>
<tr>
<td>Lexicalized (Collins)</td>
<td>87.6</td>
</tr>
<tr>
<td>Lexicalized (Charniak)</td>
<td>89.7</td>
</tr>
<tr>
<td>State of the art (2019)</td>
<td>95.6</td>
</tr>
</tbody>
</table>

The state of the art (Kitaev and Klein 2019) is obtained by using a contextual encoding of the words called BERT and a neural network for scoring the edges.
Unlike CKY, the Earley algorithm recognizes the sentence from left to right, recognizing a growing prefix.

It is an example of a top down parser (where CKY is bottom up)

Earley is mostly used in the unweighted case

Earley is generally used with grammars of variable arity (unlike CKY)
Earley for a single tree

- Earley can be viewed in the first place as a method of tree traversal:

  ![Diagram of Earley algorithm]

  where the dot is moved from node to node in the tree

- There are 3 key cases to distinguish during the traversal:
  - **predict** when the dot is before a non terminal, move it on the first immediate child
  - **scan (shift)** when the dot is before a non terminal, move it after the terminal
  - **complete** if the dot is after the last child of a rule, move the dot after the root of this rule
The Earley item is essentially a dotted rule of the form:

\[ \langle X \rightarrow \alpha \cdot Y\beta, i, j \rangle \]

where the material before the dot (\(\alpha\)) has already been recognized and the material following the dot (\(Y\beta\)) has still to be recognized.

The indexes \(i, j\) indicate the span of the recognized material on input.
Earley for CFG

We assume a sequence of words \( w_0 \ldots w_n \) and a grammar with rule set \( R \) and axiom \( S \):

- **Shift**
  \[
  \langle X \rightarrow \alpha \bullet w \beta, i, j \rangle \\
  \langle X \rightarrow \alpha w \bullet \beta, i, j + 1 \rangle \\
  \text{ (if } w = w_j \text{)}
  \]

- **Predict**
  \[
  \langle X \rightarrow \alpha \bullet Y \beta, j, j \rangle \\
  \langle Y \rightarrow \bullet \gamma, j, j \rangle \\
  \text{ (if } Y \rightarrow \gamma \in R \text{)}
  \]

- **Complete**
  \[
  \langle X \rightarrow \alpha \bullet Y \beta, i, k \rangle \\
  \langle Y \rightarrow \gamma \bullet, k, j \rangle \\
  \langle X \rightarrow \alpha Y \bullet \beta, i, j \rangle
  \]

- **Init**
  \[
  \langle \bullet S, 0, 0 \rangle
  \]

- **Goal**
  \[
  \langle S \bullet, 0, n \rangle
  \]
Ordering of inferences

- Contrary to CKY, the Earley algorithm has the **valid prefix property**, that is it the transition system naturally performs recognition from left to right in a strict incremental manner.
- Inference can be **ordered** by simply adding items to a queue called the agenda \( A \).
- Dynamic programming can be achieved by storing items in a DP table \( T \). New successor items that are already in \( T \) are not pushed anymore into the agenda.

```java
function EarleyInference(w0 ... wn, G)
    A ← \langle •S, 0, 0⟩
    T ← \{\langle •S, 0, 0⟩\}
    while A ≠ ∅ do
        item ← PopFirst(A)
        successor-set ← ApplyRules(item, w0 ... wn, G)
        PushLast(A,successor-set)
        UpdateTable(T,successor-set)
    end while
end function
```
Let's consider the tree:

```
      S
     /\  
    a   S   b
   / \  /  /
  a   b
```

The items generated to yield the successful recognition (and only those) for this tree are the following:

- \( \langle S \rightarrow \bullet a S b, 0, 0 \rangle \) (P),
- \( \langle S \rightarrow a \bullet S b, 0, 1 \rangle \) (S),
- \( \langle S \rightarrow \bullet a b, 1, 1 \rangle \) (P),
- \( \langle S \rightarrow a \bullet b, 1, 2 \rangle \) (S),
- \( \langle S \rightarrow a b \bullet, 1, 3 \rangle \) (S),
- \( \langle S \rightarrow a S \bullet b, 0, 3 \rangle \) (C),
- \( \langle S \rightarrow a S b \bullet, 0, 4 \rangle \) (S),
- \( \langle S \bullet, 0, 4 \rangle \) (C),
The earley algorithm also builds a shared forest. Shift adds the leaves of the forest and each fully completed item contributes to add a node and an hyperedge to the forest.

Example:

\[
\begin{align*}
&\langle X \rightarrow \bullet X w_4 \rangle \ (P), \ \langle X \rightarrow \bullet X w_3 \rangle \ (P), \ \langle X \rightarrow \bullet w_1 X \rangle \ (P), \ \langle X \rightarrow \bullet w_1 w_2 \rangle \ (P), \\
&\langle X \rightarrow w_1 \bullet X \rangle \ (S), \ \langle X \rightarrow w_1 \bullet w_2 \rangle \ (S), \ \langle X \rightarrow \bullet w_2 w_3 \rangle \ (S), \ \langle X \rightarrow w_1 w_2 \bullet \rangle \ (S), \\
&\langle X \rightarrow w_2 \bullet w_3 \rangle \ (S), \ \langle X \rightarrow X \bullet w_3 \rangle \ (C), \ \langle X \rightarrow w_2 w_3 \bullet \rangle \ (S), \ \langle X \rightarrow X w_3 \bullet \rangle \ (S), \\
&\langle X \rightarrow w_1 X \bullet \rangle \ (C), \ \langle X \rightarrow X \bullet w_4 \rangle \ (C), \ \langle X \rightarrow X w_4 \bullet \rangle \ (S), \ \langle X \bullet \rangle \ (C)
\end{align*}
\]
Plan

1. Context free grammar parsing algorithms
2. Transition based parsing for dependency grammar
3. Dynamic programming for dependency grammar
4. CKY for tree adjoining grammars
Shift reduce parsing

- **Shift reduce** parsing for **projective** dependency grammar is a popular method inherited from LR parsing (Knuth 1965) for non ambiguous grammars.
- In NLP the shift reduce conflicts are resolved by means of a machine learning based scoring function.
- The method is very popular and mostly used with approximative search methods such as greedy or beam search.
- The system can be understood as a form of PDA whose parse state or **configuration** is described by a triple:

\[ \langle S, B, A \rangle \]

where the stack **S** is filled with word symbols, the buffer **B** is made of not yet processed words and **A** is a set of dependency arcs (couples of words)
The init, end states and transitions of the automaton are the following:

- **init** \( \langle \epsilon, w_0 \ldots w_n, \emptyset \rangle \)
- **goal** \( \langle w_0, \epsilon, A \rangle \)
- **shift** \( \langle S, w_i | B, A \rangle \Rightarrow \langle S | w_i, B, A \rangle \)
- **left – arc** \( \langle S | w_i | w_j, B, A \rangle \Rightarrow \langle S | w_j, B, A \cup \{ j \rightarrow i \} \rangle \) \((i \neq w_0)\)
- **right – arc** \( \langle S | w_i | w_j, B, A \rangle \Rightarrow \langle S | w_i, B, A \cup \{ i \rightarrow j \} \rangle \)

This transition system is called **arc-standard**, others are possible.
Example

- J'ai réservé un vol pour Sophie
- Reference parse :

```plaintext
root j' ai réservé un vol pour Sophie
```
## Example

<table>
<thead>
<tr>
<th><strong>ARCS</strong></th>
<th><strong>STACK</strong></th>
<th><strong>BUFFER</strong></th>
<th><strong>ACTION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root</td>
<td>j’, ai, réservé, un, vol, pour, Sophie</td>
<td>shift</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, j’</td>
<td>ai, réservé, un, vol, pour, Sophie</td>
<td>shift</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, j’, ai</td>
<td>réservé, un, vol, pour, Sophie</td>
<td>shift</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, j’, ai, réservé</td>
<td>un, vol, pour, Sophie</td>
<td>left-arc</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, j’, réservé</td>
<td>un, vol, pour, Sophie</td>
<td>left-arc</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé</td>
<td>un, vol, pour, Sophie</td>
<td>shift</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé, un</td>
<td>vol, pour, Sophie</td>
<td>shift</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé, un, vol</td>
<td>pour, Sophie</td>
<td>left-arc</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé, vol</td>
<td>pour, Sophie</td>
<td>right-arc</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé</td>
<td>pour, Sophie</td>
<td>shift</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>ARCS</th>
<th>STACK</th>
<th>BUFFER</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé, pour</td>
<td>Sophie</td>
<td>shift</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé, pour, Sophie</td>
<td>ε</td>
<td>right-arc</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé, pour</td>
<td>ε</td>
<td>right-arc</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root, réservé</td>
<td>ε</td>
<td>right-arc</td>
</tr>
<tr>
<td>root j’ ai réservé un vol pour Sophie</td>
<td>root</td>
<td>ε</td>
<td>goal!</td>
</tr>
</tbody>
</table>
Greedy search with Arc standard

function ArcStandardGreedyParse(w_0 \ldots w_n)
    \( c \leftarrow \langle w_0, w_1 \ldots w_n, \emptyset \rangle \)
    while \( c \) is not goal do
        \( a \leftarrow \text{argmax}_{a \in A} \psi(a, c) \)
        \( c \leftarrow a(c) \)
    end while
    return \( c \)
end function

Ends with \( c = \langle S, B, A \rangle \), where \( A \) is the dependency tree.

Too naïve?

- Arc standard is often used with approximative search methods, greedy or beam search
- Algorithm with local decisions used in a deep learning context (≈ state of the art if \( \psi \) is well informed, which is the case if a BI-LSTM encodes the words (class 2)).
Plan

1. Context free grammar parsing algorithms
2. Transition based parsing for dependency grammar
3. Dynamic programming for dependency grammar
4. CKY for tree adjoining grammars
We suppose that words are indexed by integers. Intuitively we can represent a dependency tree spanning words from $i$ to $j$ with root $h$ by a triple $\langle h, i, j \rangle$.

For instance $\langle 2, 0, 5 \rangle$ is the item that covers the full sentence with head at position 2:

\[
\begin{array}{cccccc}
D & N & V & P & D & N \\
the & cat & sleeps & on & the & mat \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

and $\langle 3, 3, 5 \rangle$ is the item that covers the subsentence *on the mat* with root *on*.
Naive CKY

- Naïve CKY for projective dependency grammars uses essentially the following two reduction rules:

\[
\text{LEFT-REDUCE} = \frac{\langle h_l, i, k \rangle \langle h_r, k, j \rangle}{\langle h_l, i, j \rangle}
\]

\[
\text{RIGHT-REDUCE} = \frac{\langle h_l, i, k \rangle \langle h_r, k, j \rangle}{\langle h_r, i, j \rangle}
\]

this amounts to link iteratively the roots of contiguous spans

Observation on complexity

The complexity of this algorithm is in $\mathcal{O}(n^5)$ where the complexity for the PCFG version is in $\mathcal{O}(n^3)$. 
Example of scoring function

- As in the lexicalized case, the scoring function is typically a composition of potentials $\psi(e, x)$ scored on the hyperedges.
- In the dependency case, an hyperedge takes the following form:

$$\langle h_{\{l, r\}}, i, j \rangle, \langle h_l, i, k \rangle \langle h_r, k, j \rangle$$

where the $\{l, r\}$ is dependant of the reduction rule used (left or right)

The scoring function can evaluate the relationship between a head (e.g. *mange*) and its dependant (e.g. *couvert, tomate*) together with the surrounding words in the sentence.
We consider the example:

\[
\begin{array}{ccccccc}
D & N & V & P & D & N \\
the & cat & sleeps & on & the & mat \\
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

where we build the items leading to the solution following a topological order on the forest:

\[
\begin{align*}
&\langle cat, the, cat \rangle \ (RR) \\
&\langle sleeps, the, sleeps \rangle \ (RR) \\
&\langle mat, the, mat \rangle \ (RR) \\
&\langle on, on, mat \rangle \ (LR) \\
&\langle sleeps, sleeps, mat \rangle \ (LR) \\
\end{align*}
\]

where we replaced integer indexes with their corresponding strings for readability.
Idea of Eisner’s algorithm

- Eisner’s version of CKY for dependency grammar uses spans as basic representation, and constrains heads to be on the edges of the spans (hence reducing the combinatorics). In other words, the item gets the following representation:

  \[
  \langle i, j, \text{dir}, c \rangle
  \]

  where \( i, j \) are the indexes of the span, \( \text{dir} \in \{\leftarrow, \rightarrow\} \) indicates the position of the children wrt the head, and \( c \in \{\circ, \bullet\} \) indicates if the item is complete or not.

- Inference in Eisner’s formulation amounts to deduce larger and larger spans until a span covers the whole sentence. The ordering of inferences follows that of traditional CKY.

- Eisner’s representation manages to get an inference in \( O(n^3) \).
Inference rules

There are two ways to create larger spans from smaller spans:

- **By adding** an edge linking both subspans heads:

\[
\begin{array}{c}
h \quad r \\
\end{array}
\quad \Rightarrow 
\begin{array}{c}
h \\
\end{array} \quad \begin{array}{c}
r + 1 \\
m \end{array}
\]

Creates an incomplete span from two completed spans

- **By concatenating** two spans without adding any edge:

\[
\begin{array}{c}
h \quad m \\
\end{array}
\quad \Rightarrow 
\begin{array}{c}
h \\
\end{array} \quad \begin{array}{c}
m \\
e \end{array}
\]

Creates a complete span from an incomplete and a complete span
Inference rules
full version

By considering the two directions (left and right) we get the following (full) set of inference rules:

- Linking rules:
  \[ \text{RIGHT-LINK}(RL) = \frac{\langle h, r, \rightarrow, \bullet \rangle \langle r + 1, m, \leftarrow, \bullet \rangle}{\langle h, m, \rightarrow, \circ \rangle} \]
  \[ \text{LEFT-LINK}(LL) = \frac{\langle m, r, \rightarrow, \bullet \rangle \langle r + 1, h, \leftarrow, \bullet \rangle}{\langle m, h, \leftarrow, \circ \rangle} \]

- Completion rules:
  \[ \text{RIGHT-COMPLETE}(RC) = \frac{\langle h, m, \rightarrow, \circ \rangle \langle m, e, \rightarrow, \bullet \rangle}{\langle h, e, \rightarrow, \bullet \rangle} \]
  \[ \text{LEFT-COMPLETE}(LC) = \frac{\langle e, m, \leftarrow, \circ \rangle \langle m, h, \leftarrow, \bullet \rangle}{\langle e, h, \leftarrow, \bullet \rangle} \]

- At init, each word \( w_i \) introduces its own completed spans:
  \[ \langle i, i, \leftarrow, \bullet \rangle \quad \langle i, i, \rightarrow, \bullet \rangle \]
Example

Let’s see how Eisner’s derives *the cat sleeps on the mat*:

```
root  the  cat  sleeps  on  the  mat
-1    0     1     2     3     4     5
```

we add a dummy root since we need a head at the edge of a span at the end

- **Items of length 0**: \( \langle i, i \rightarrow, \bullet \rangle \langle i, i \leftarrow, \bullet \rangle \) (1 ≤ i ≤ 5)
- **Items of length 1**: \( \langle 0, 1 \leftarrow, \circ \rangle \) (LL), \( \langle 1, 2 \leftarrow, \circ \rangle \) (LL), \( \langle 2, 3 \rightarrow, \circ \rangle \) (RL), \( \langle 4, 5 \leftarrow, \circ \rangle \) (RL), \( \langle 0, 1 \leftarrow, \bullet \rangle \) (LC), \( \langle 4, 5 \leftarrow, \bullet \rangle \) (RC)
- **Items of length 2**: \( \langle 3, 5 \rightarrow, \circ \rangle \) (RL), \( \langle 0, 2 \leftarrow, \bullet \rangle \) (LC), \( \langle 3, 5 \rightarrow, \bullet \rangle \) (RC)
- **Items of length 3**: \( \langle -1, 2, \rightarrow, \circ \rangle \) (RL), \( \langle 2, 5 \rightarrow, \bullet \rangle \) (RC)
- **Items of length 6**: \( \langle -1, 5, \rightarrow, \bullet \rangle \) (RC)

**Only the items leading the to solution are given (!)**
Plan

1. Context free grammar parsing algorithms
2. Transition based parsing for dependency grammar
3. Dynamic programming for dependency grammar
4. CKY for tree adjoining grammars
CKY for tree adjoining grammars

- This algorithm provides an example for parsing mildly context sensitive languages.
- It highlights the relation between gap processing and parsing of mildly context sensitive systems.
- The algorithm is restricted to trees of arity at most 2.

Reminder on context sensitivity and gaps

Context sensitive patterns are generated by grammars involving wrapping adjunction:

\[
\begin{align*}
S & \quad S_{na} \\
\epsilon & \quad a & S & \quad d \\
\downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
b & \quad S_{\star na} & c \\
\end{align*}
\]

\[
\begin{align*}
S & \quad S_{na} & \quad S_{na} \\
\epsilon & \quad a & \quad S & \quad b \\
\downarrow & \quad \downarrow & \quad \downarrow & \quad \downarrow \\
S_{\star na} & \quad a & \quad S_{\star na} & \quad b \\
\end{align*}
\]
The wrapping auxiliary tree and the discontinuity

- The wrapping auxiliary tree covers a discontinuous span:

- Simple example:
The **CKY** algorithm and its item

- Informally, **CKY** moves dots from word leaves along the trees until it reaches the root of the tree. Below the dot, the tree structure is already recognized, above the dot the tree structure is not yet recognized:

![Diagram showing a tree with dots moving from leaves to the root.]

- To describe a parse state we use the following item:

$$\langle \alpha, p_t^{(i)}, i, f_L, f_R, j \rangle$$

$\alpha$ is the tree identifier, $p$ is the address of the tree node, $t \in \{\top, \bot\}$ indicates whether the dot is above or below the node, the optional $(i)$ indicates the $i$th child of node $p$ and finally $i, j, f_L, f_R$ are span indexes used as in **CFG** algorithms described earlier. They indicate here the span covered by the current position of the parser.
Internal tree inferences

- **Lex scan:**

\[
\langle \alpha, p^{\top}_T, i, -, -, i + 1 \rangle \quad \text{(label}(\alpha, p) = w_i)\]

Inserts tree \(\alpha\) anchored by word \(w_i\). '-' means 'no gap'.

- **Move up unary:**

\[
\frac{\langle \alpha, p^{(1)}_\top, i, f_L, f_R, j \rangle}{\langle \alpha, p_\bot, i, f_L, f_R, j \rangle}
\]

Moves the dot from the top of the first (and unique) child to the bottom of its parent.

- **Move up binary:**

\[
\frac{\langle \alpha, p^{(1)}_\top, i, f_L, f_R, k \rangle \quad \langle \alpha, p^{(2)}_\top, k, f'_L, f'_R, j \rangle}{\langle \alpha, p_\bot, i, f_L \otimes f'_L, f_R \otimes f'_R, j \rangle}
\]

Moves the dots from the top of the two children up to the bottom of its parent, where:

\[
f \otimes g = \begin{cases} 
  f & \text{if } g = ' -' \\
  g & \text{if } f = ' -' \\
  \bot & \text{otherwise (error)}
\end{cases}
\]
Substitution

- **Substitution** amounts to move the dot from the top of tree $\alpha$ to the top of a substitution node in tree $\gamma$:

\[
\frac{\langle \alpha, p_T, i, -, -, j \rangle}{\langle \gamma, p_T, i, -, -, j \rangle} \quad (\text{label}(\alpha, p) = \text{label}(\gamma, p), \text{root}(\alpha, p))
\]

The dot moves immediately to top because no adjunction is allowed on substitution nodes.
Adjunction (overview)

Adjunction is expressed by three rules:

- **Foot-predict** that launches the recognition of an auxiliary tree from the bottom of a node $n$
- **Adjoin** that finishes the adjunction on the top of $n$
- **No adjoin** that just moves the dot from bottom to top on the node $n$ (and does not perform any adjunction at all):

$$\langle \alpha, p_\perp, i, f_L, f_R, j \rangle$$

$$\langle \alpha, p_\top, i, f_L, f_R, j \rangle$$
Foot predict

Foot predict is defined as:

\[
\frac{\langle \alpha, p_\bot, i, -, -, j \rangle}{\langle \beta, p_\top, i, i, j, j \rangle} \quad (\text{label}(\alpha, p) = \text{label}(\beta, p))
\]

This inserts the foot node of the auxiliary tree \( \beta \) with gap \( i, j \), that is the span covered by \( \text{node}(\alpha, p) \).
Adjoin finishes an adjunction launched by foot predict by moving the dot from top to bottom on \( p \) in \( \alpha \). The operation checks that \( \alpha \) fills the span \((f_L, f_R)\) that is not covered by \( \beta \)

\[
\langle \beta, p^\top, i, f_L, f_R, j \rangle \langle \alpha, p^\bot, f_L, f_L', f_R', f_R \rangle \\
\langle \alpha, p^\top, i, f_L', f_R', j \rangle
\]
The CKY algorithm has a complexity in $\mathcal{O}(n^6)$. The bottleneck is located at adjoin that has 6 free variables ranging over $n$.

- Mildly context sensitive variants of TAG have complexities of the form $\mathcal{O}(n^{3f})$ where $f$ is the fan-out of the system. TAG has fan-out 2 because it generates two discontinuous spans (and one gap between).
We parse $axb$ with the grammar:

$$\alpha_1: A \quad \alpha_2: B \quad \alpha_0: S \quad \beta: X$$

We use node labels as addresses to simplify notations.
Structured models are in principle theoretically motivated and good models of language.

The level of ambiguity at real scale requires to use disambiguation methods.

The machine learning methods for disambiguation require annotated data (such as treebanks).

This is a serious limitation: research efforts are dedicated to minimize the amount of supervision, for instance by learning to perform some sort of inference for which supervision is cheap rather than just predict the structure (see readings).