# Dependency Parsing

Computational Linguistics

Alexander Koller

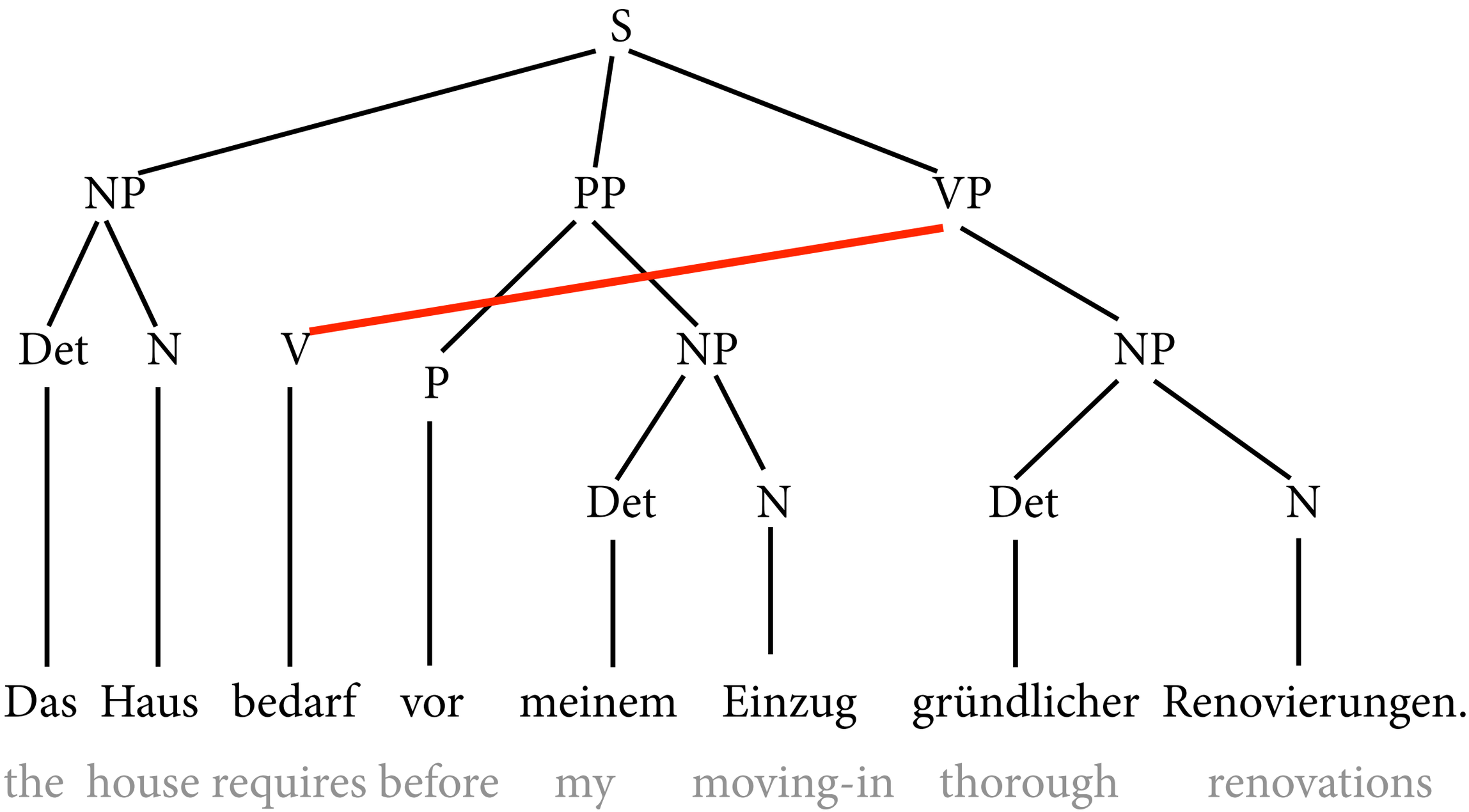
14 December 2018

# Discontinuous constituents

- So far, we have talked about *phrase-structure* parsing.
  - substrings form constituents of various syntactic categories
  - every constituent must be a contiguous substring

- This assumption mostly correct for English.
  For other languages, it doesn't work so well.

# Example

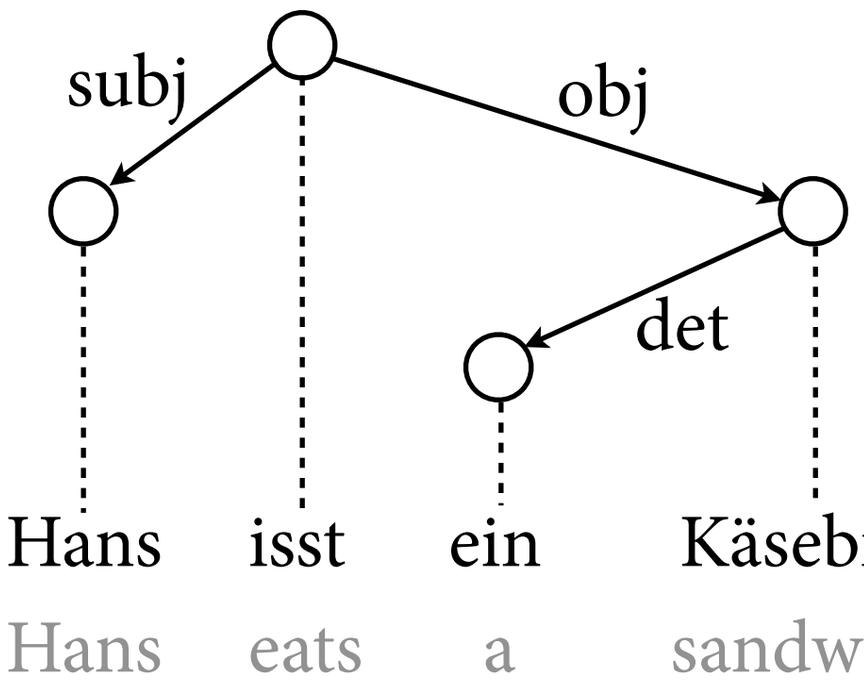

S
NP
PP
VP
Det
N
V
P
NP
Det
N
NP
Det
N
Das Haus bedarf vor meinem Einzug gründlicher Renovierungen.
the house requires before my moving-in thorough renovations

# Dependency trees

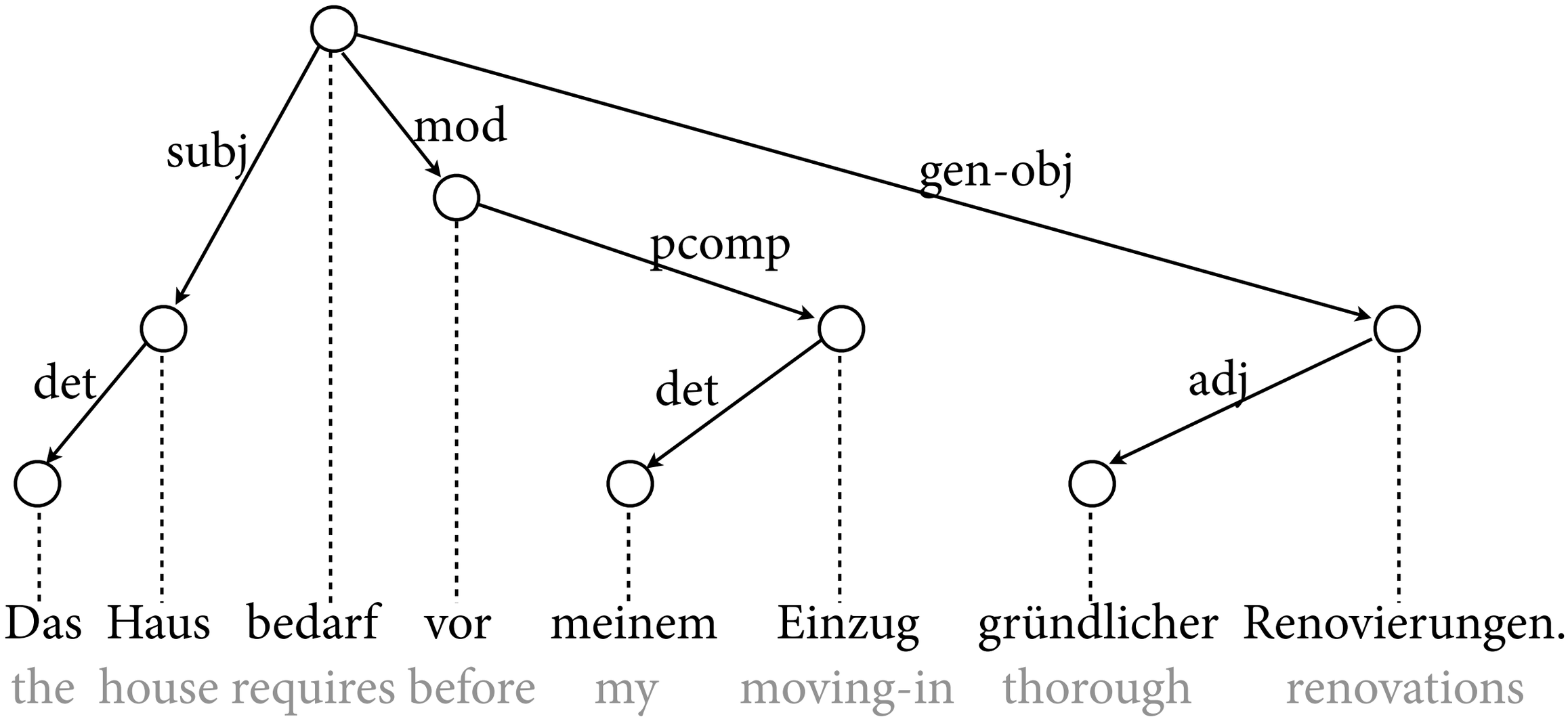
- Basic idea:
  - no constituents, just relations between words
  - nodes of tree = words; edges = relations
  - grammar specifies valency of each word

- Brief history:
  - Tesniere 1953, posthumously
  - Prague School during Cold War
  - very important in CL since 2005 or so (Nivre, McDonald)

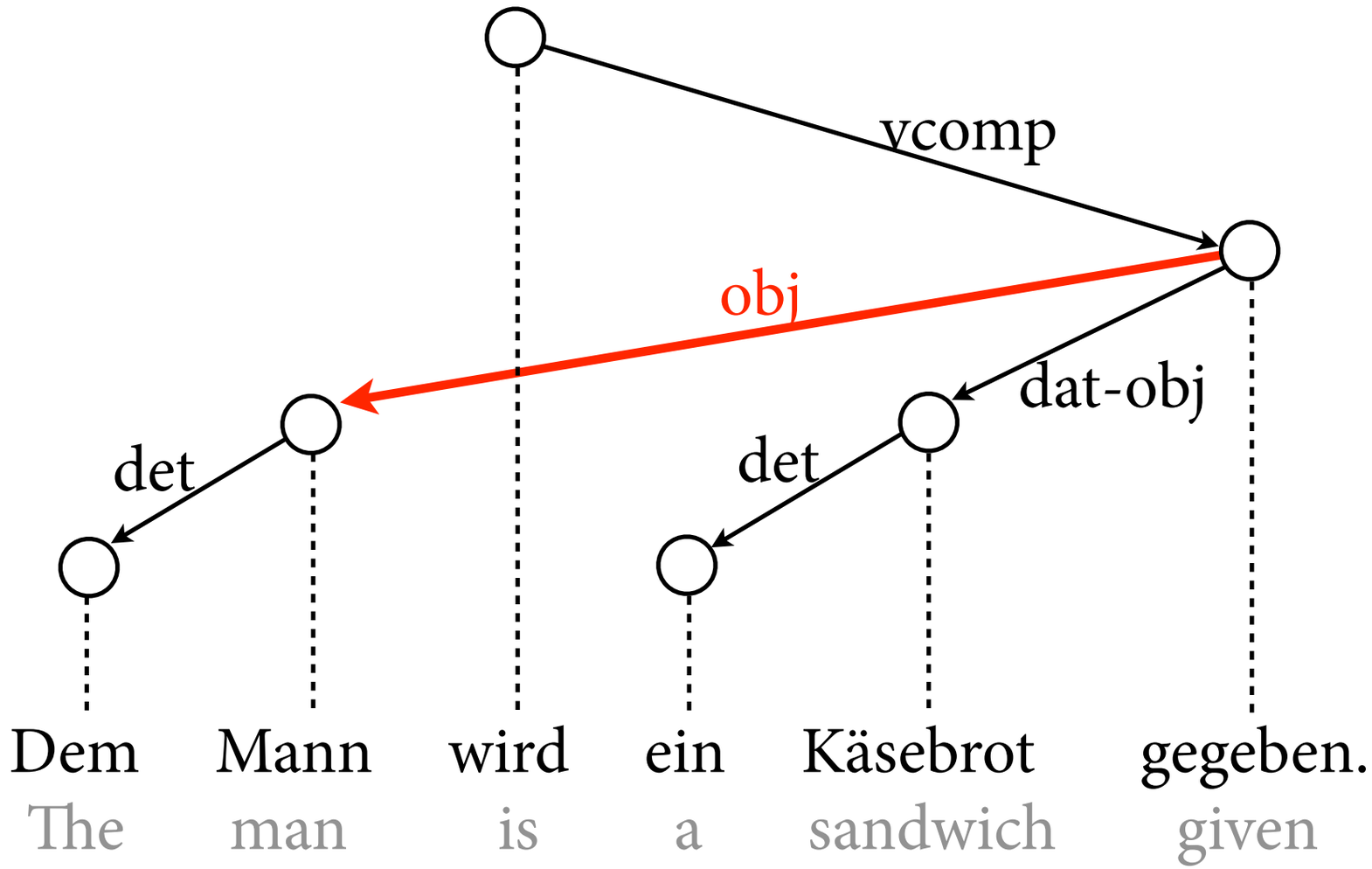
# A dependency tree



# A dependency tree

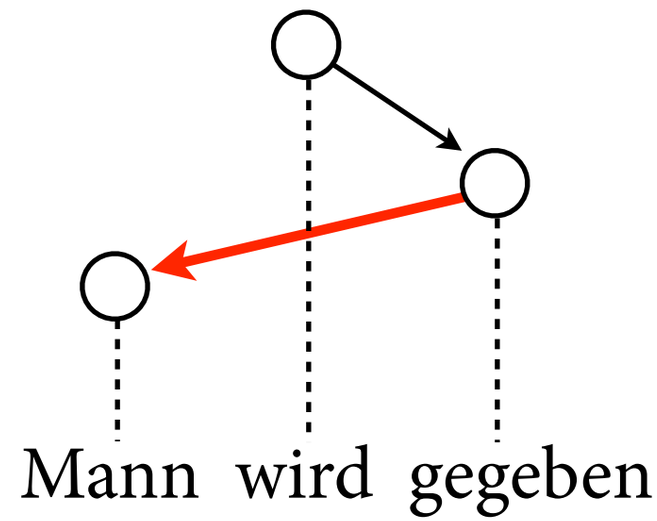


# A dependency tree



# Projectivity

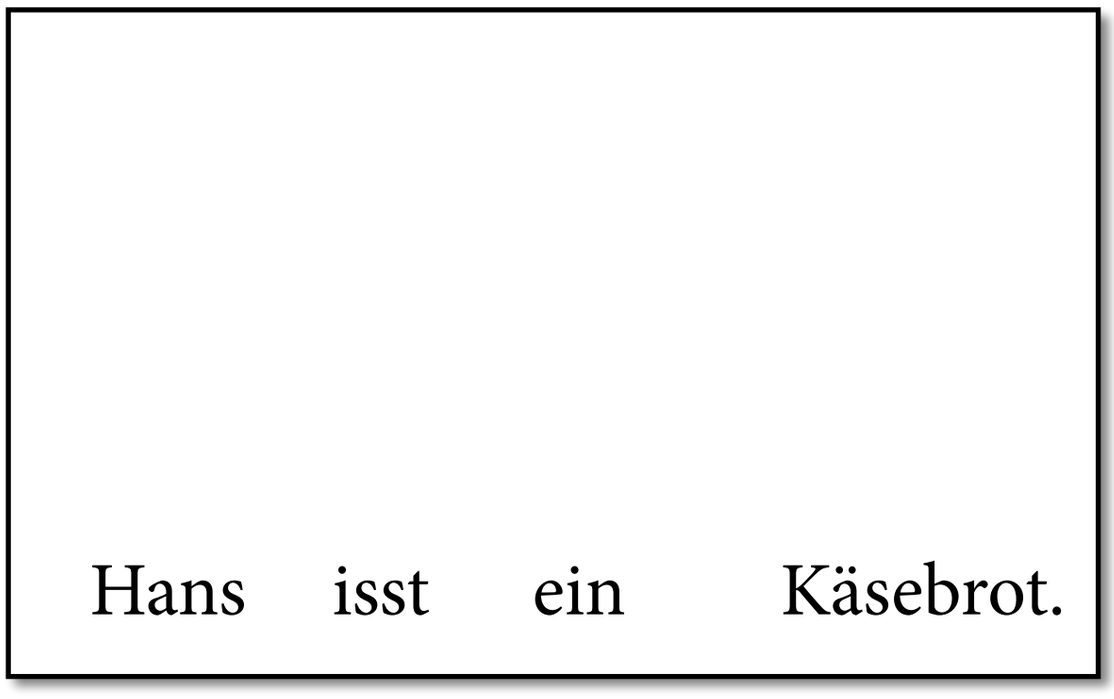
- Dependency tree may have *crossing edges*, which cross the *projection line* of another word.




- A dependency tree is called *projective* iff it has no crossing edges.

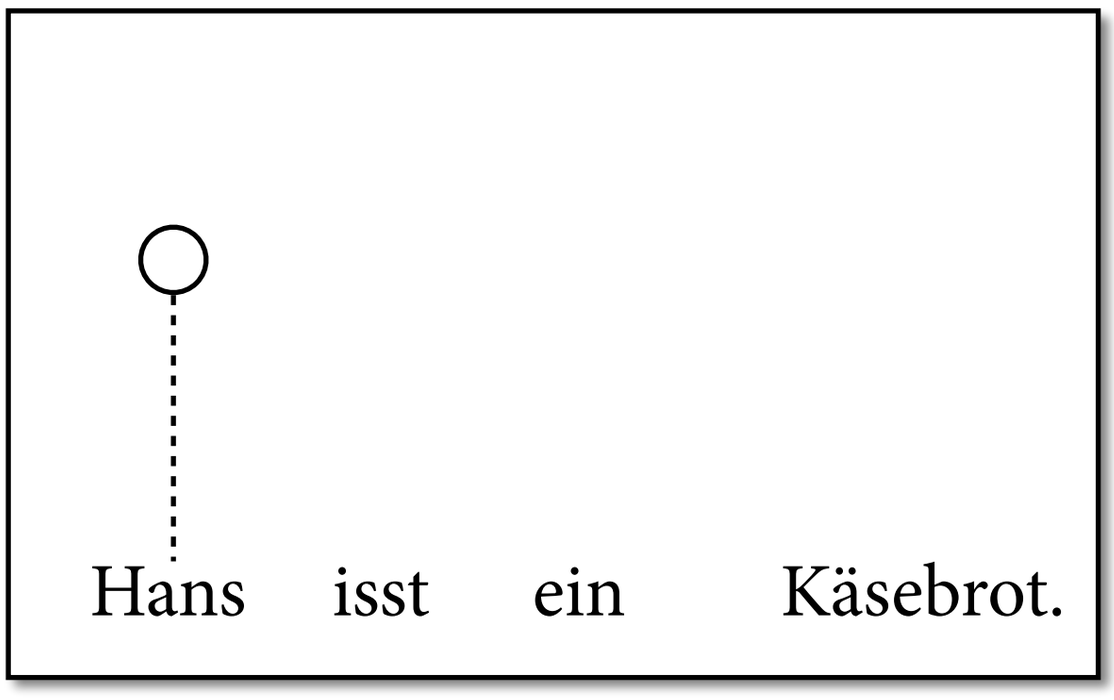
# Nivre-style dependency parsing


- Idea by Joakim Nivre (2003):
  - read sentence word by word, left to right
  - after each word, select a *parser operation* from large set by consulting a machine-learned classifier
  - original algorithm constructs only projective trees; can be extended to non-projective parsing too



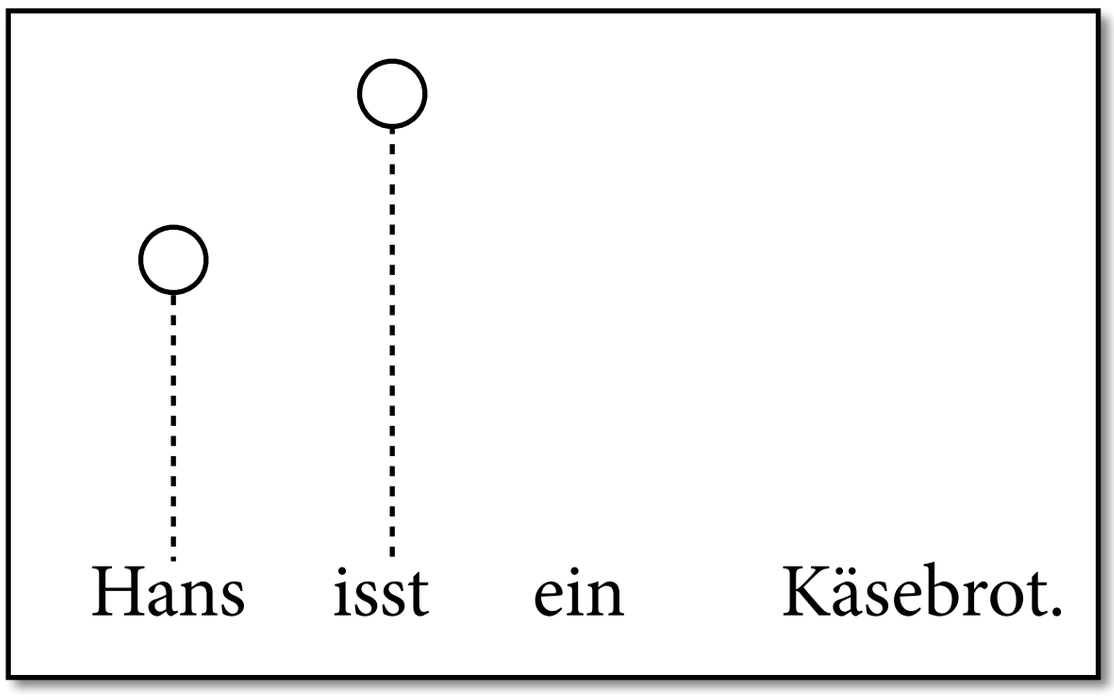
# Nivre-style dependency parsing


- Idea by Joakim Nivre (2003):
  - read sentence word by word, left to right
  - after each word, select a *parser operation* from large set by consulting a machine-learned classifier
  - original algorithm constructs only projective trees; can be extended to non-projective parsing too



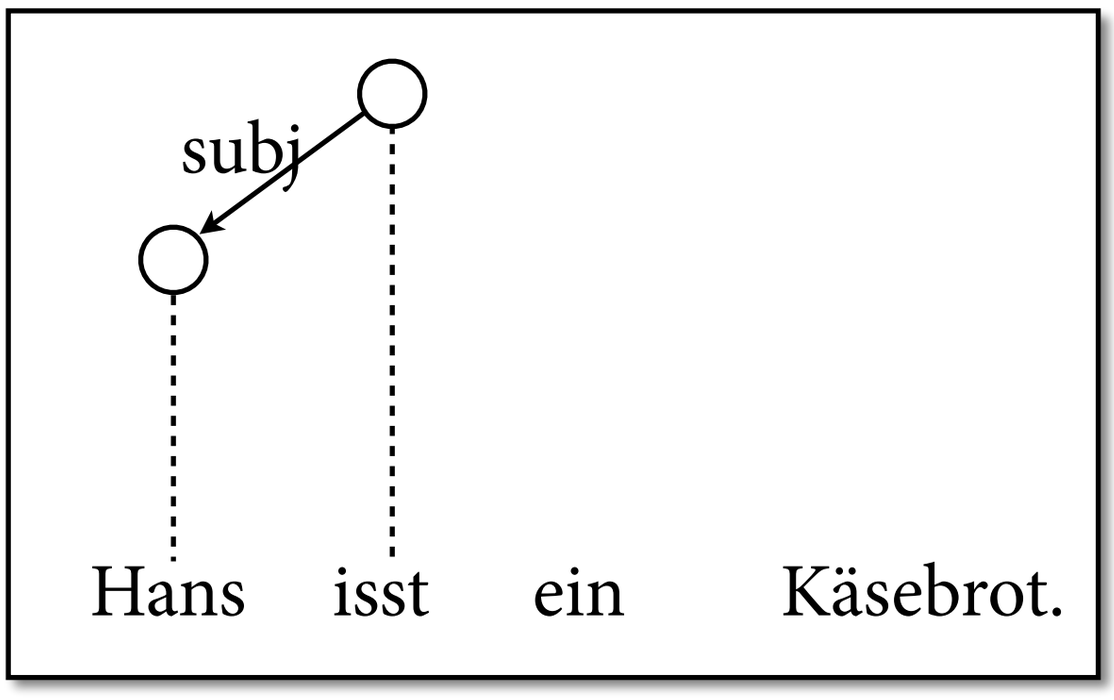
# Nivre-style dependency parsing


- Idea by Joakim Nivre (2003):
  - read sentence word by word, left to right
  - after each word, select a *parser operation* from large set by consulting a machine-learned classifier
  - original algorithm constructs only projective trees; can be extended to non-projective parsing too



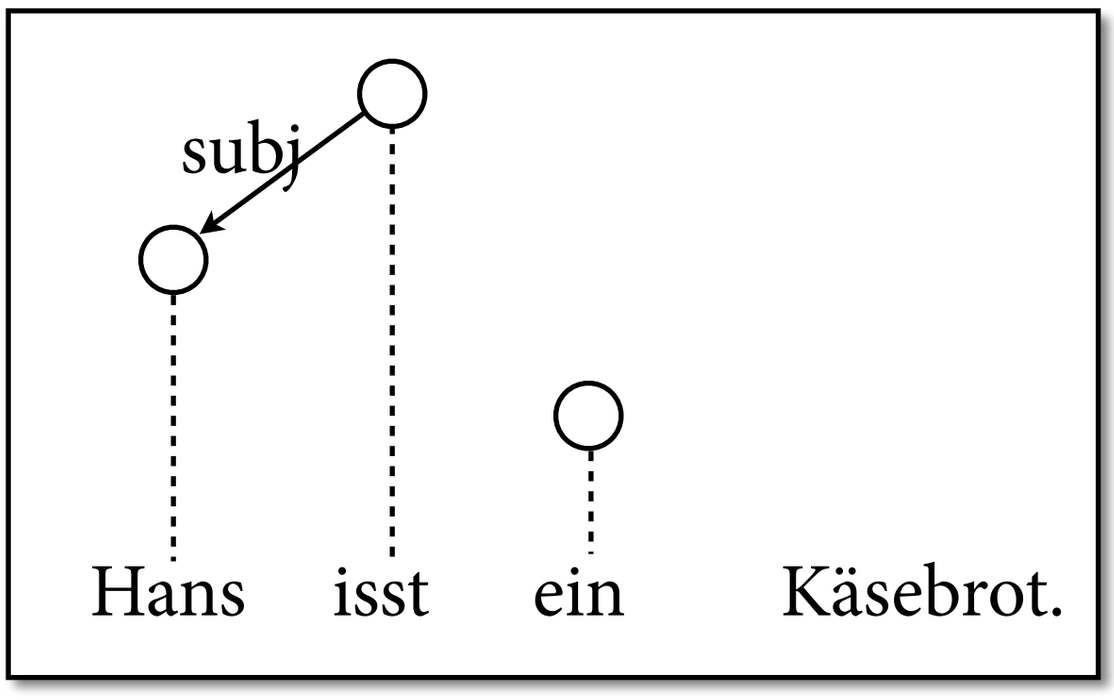
# Nivre-style dependency parsing


- Idea by Joakim Nivre (2003):
  - read sentence word by word, left to right
  - after each word, select a *parser operation* from large set by consulting a machine-learned classifier
  - original algorithm constructs only projective trees; can be extended to non-projective parsing too



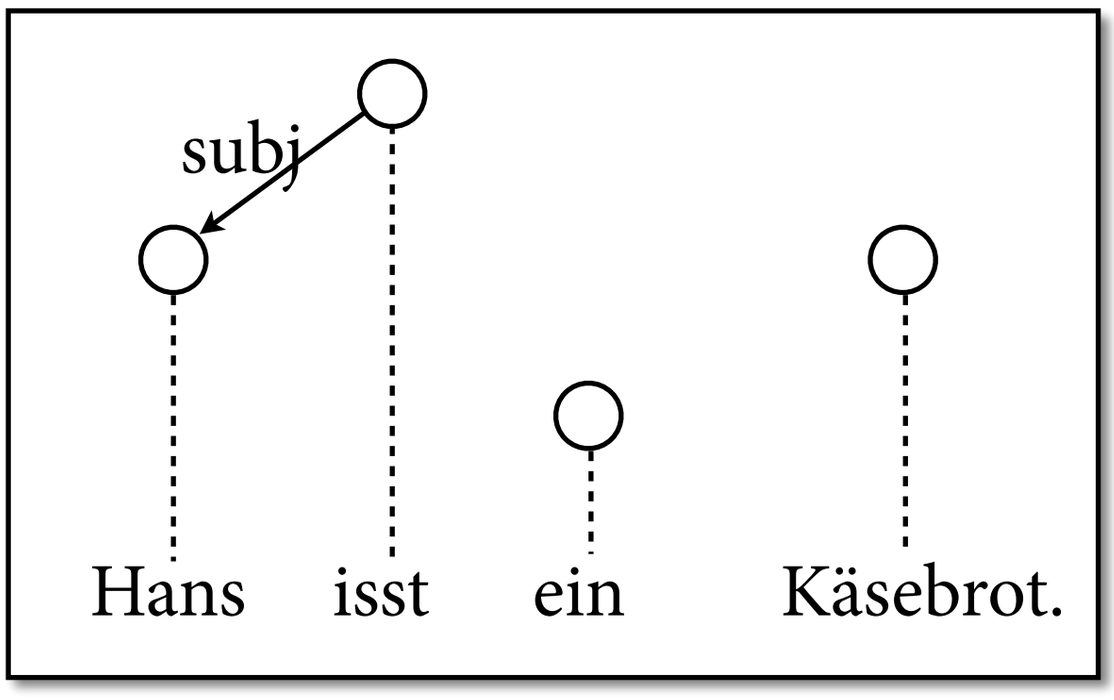
# Nivre-style dependency parsing


- Idea by Joakim Nivre (2003):
  - read sentence word by word, left to right
  - after each word, select a *parser operation* from large set by consulting a machine-learned classifier
  - original algorithm constructs only projective trees; can be extended to non-projective parsing too



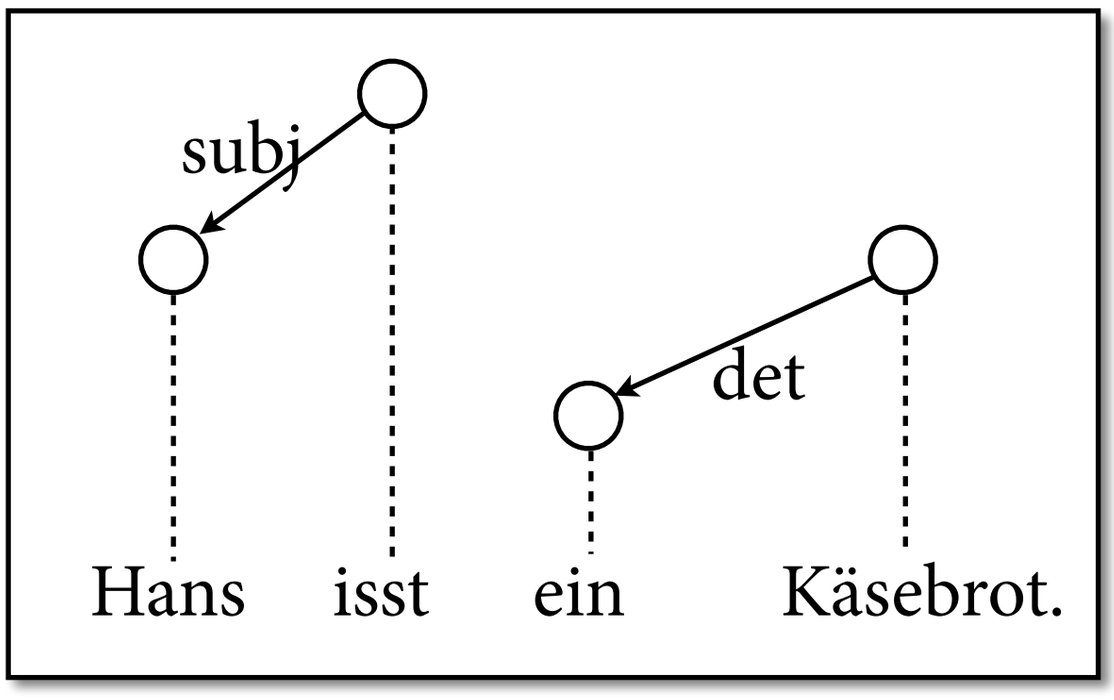
# Nivre-style dependency parsing


- Idea by Joakim Nivre (2003):
  - read sentence word by word, left to right
  - after each word, select a *parser operation* from large set by consulting a machine-learned classifier
  - original algorithm constructs only projective trees; can be extended to non-projective parsing too



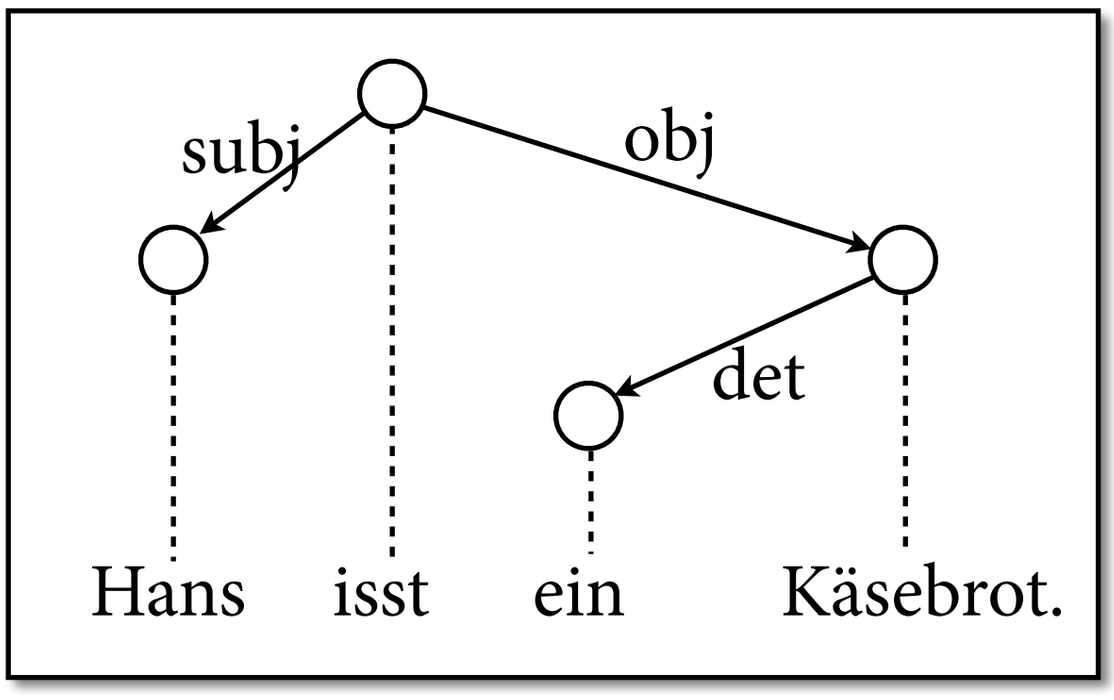
# Nivre-style dependency parsing


- Idea by Joakim Nivre (2003):
  - read sentence word by word, left to right
  - after each word, select a *parser operation* from large set by consulting a machine-learned classifier
  - original algorithm constructs only projective trees; can be extended to non-projective parsing too



# Nivre-style dependency parsing

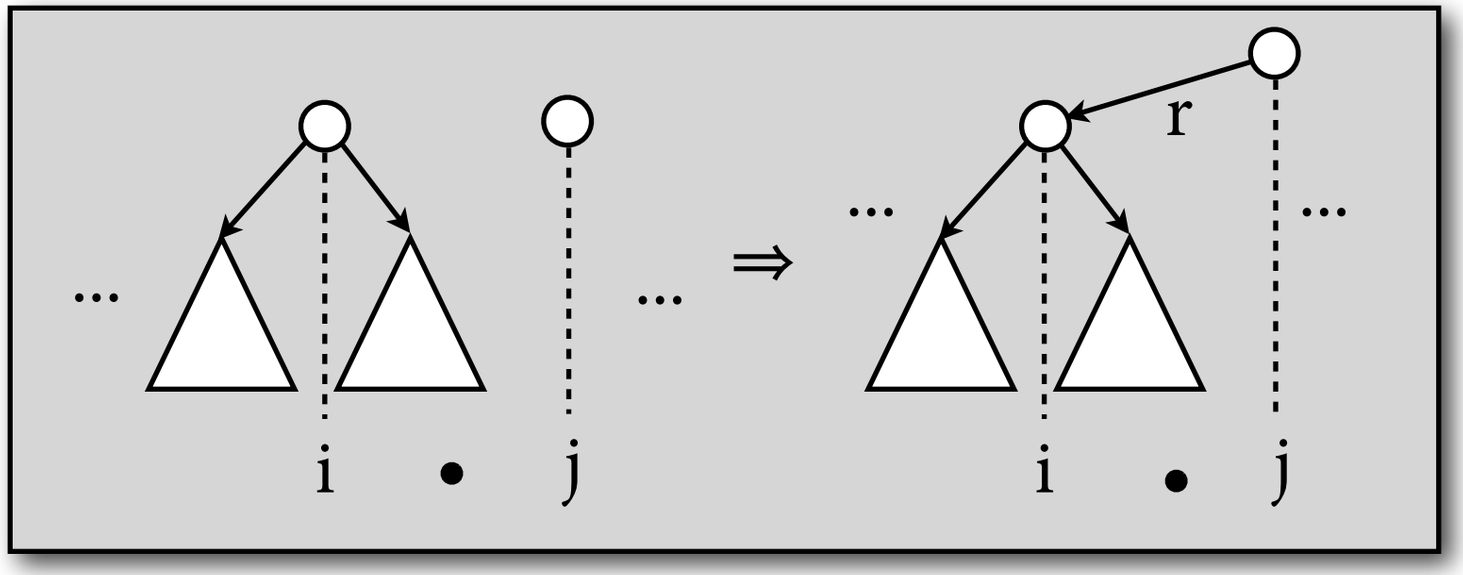

- Idea by Joakim Nivre (2003):
  - read sentence word by word, left to right
  - after each word, select a *parser operation* from large set by consulting a machine-learned classifier
  - original algorithm constructs only projective trees; can be extended to non-projective parsing too



# Left-Arc operation

- Left-Arc(r): Topmost token *i* on stack becomes left r-child of next input token *j*.

$$\frac{(\sigma \cdot i,\ j \cdot \tau,\quad \mathrm{h},\ \mathrm{d}) \quad \mathrm{h}(i) = 0}{(\sigma,\quad j \cdot \tau,\quad \mathrm{h}[i \mapsto j],\ \mathrm{d}[i \mapsto \mathrm{r}])}$$

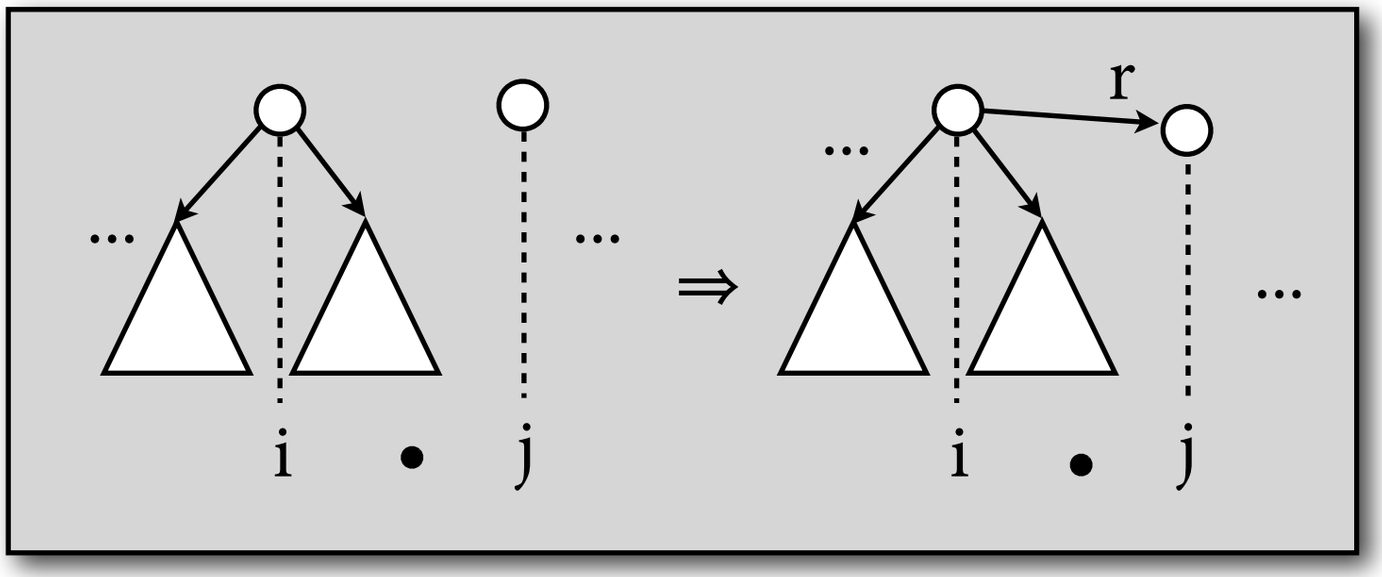



- *i* disappears from stack, because *i* can't get further children in a projective tree

# Right-Arc operation

- Right-Arc(r): Input token *j* becomes (right) r-child of topmost stack token *i*.

$$\frac{(\sigma \cdot i, \quad j \cdot \tau, \quad \mathrm{h}, \mathrm{d}) \quad \mathrm{h}(j) = 0}{(\sigma \cdot i \cdot j, \ \tau, \quad \mathrm{h}[j \mapsto i], \mathrm{d}[j \mapsto \mathrm{r}])}$$

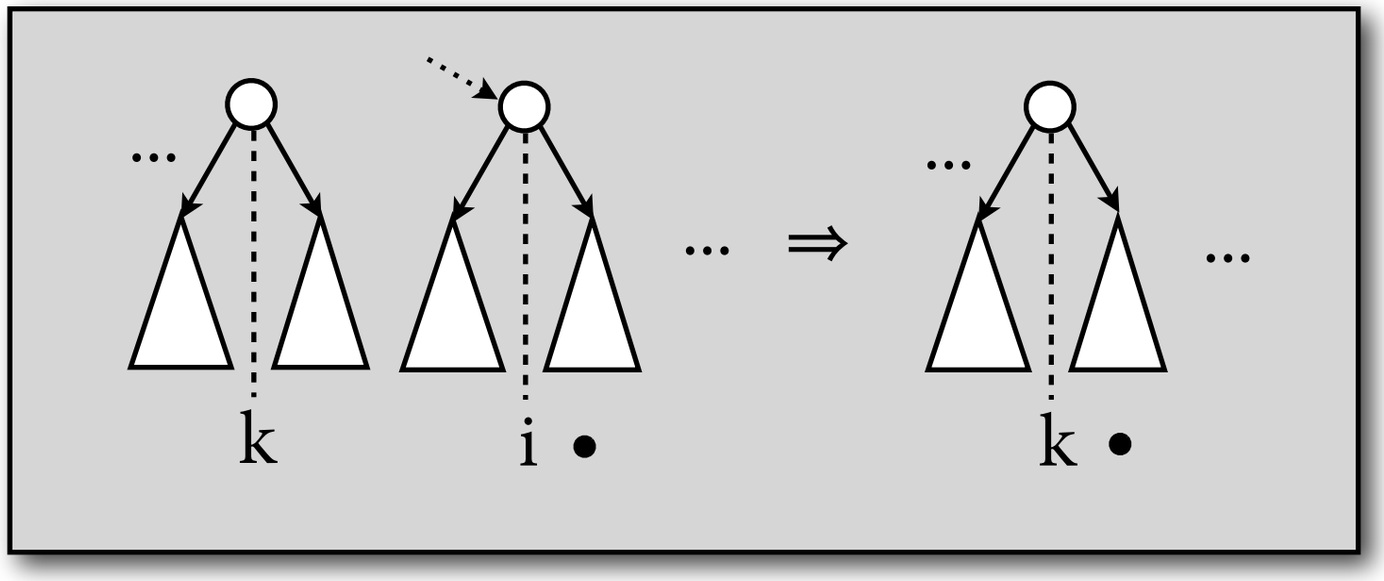



- *i*, *j* both remain on stack because they can receive further children (on the right).

# Reduce operation

- Reduce: Remove topmost token from stack.

$$\frac{(\sigma \cdot i,\ \tau,\ \mathrm{h},\ \mathrm{d}) \quad \mathrm{h}(i) \neq 0}{(\sigma,\ \tau,\ \mathrm{h},\ \mathrm{d})}$$



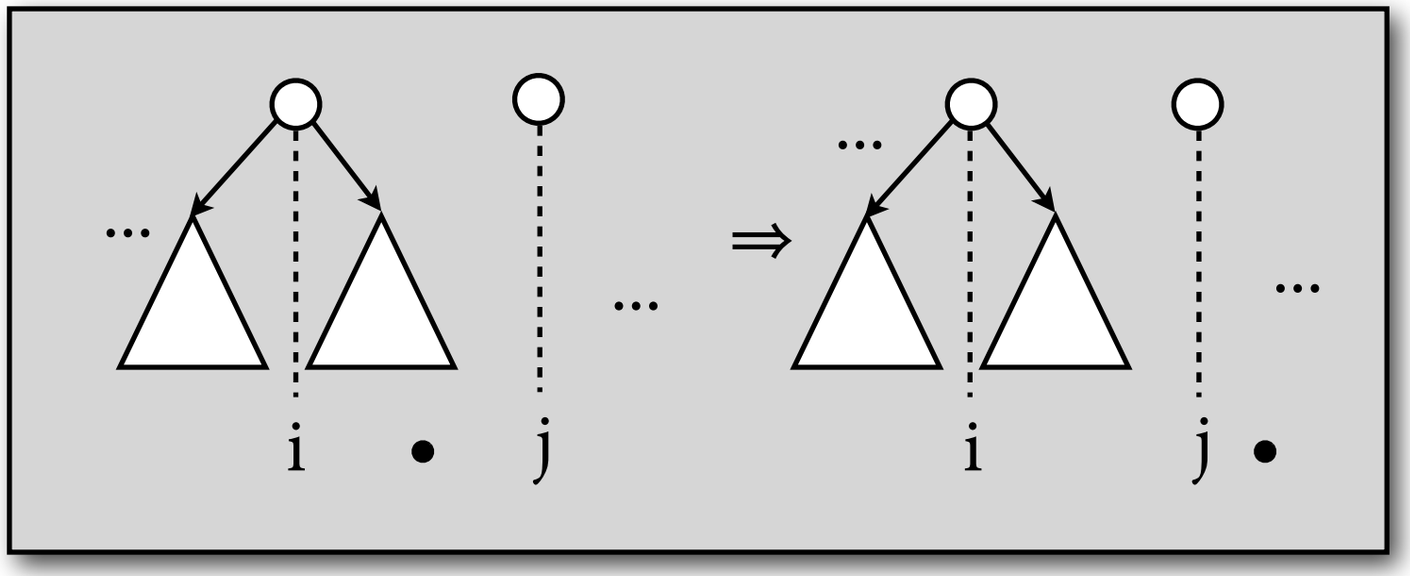

- This decides that we have seen all children of *i*, and makes words further to the left available for receiving further right children.

- Rule requires that *i* already has a parent.

# Shift operation

- Shift: Moves next input token *j* to stack.

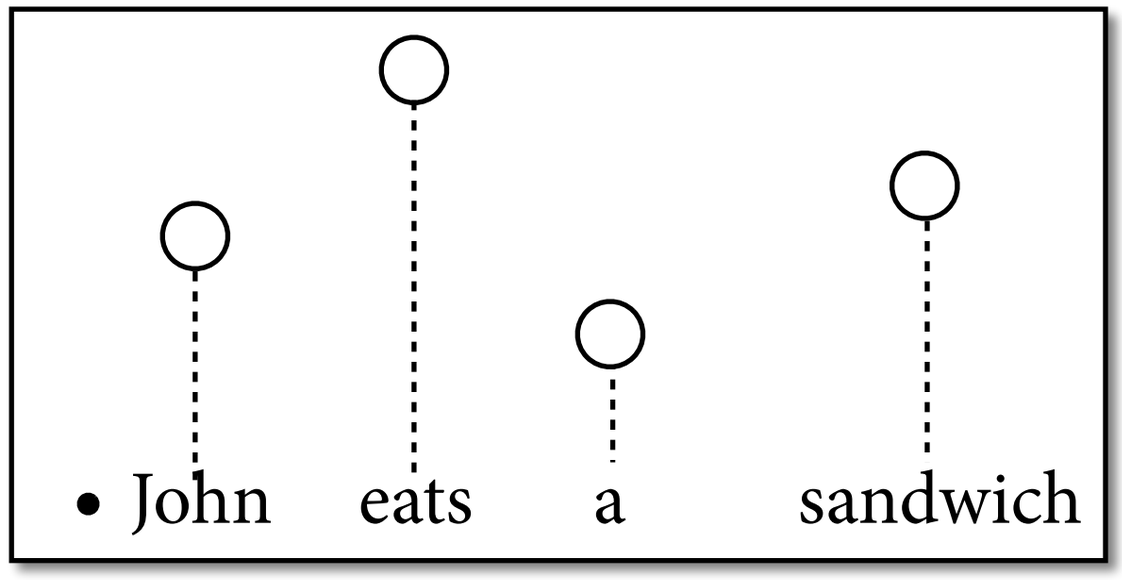
$$\frac{(\sigma,\ \ j\cdot\tau,\ \ h, d)}{(\sigma\cdot j,\ \tau,\ \ h, d)}$$




- Decides that *j* and any word *i* on stack are in disjoint tree positions.

# Example run

(ε, J eats a sw)




John eats a sandwich

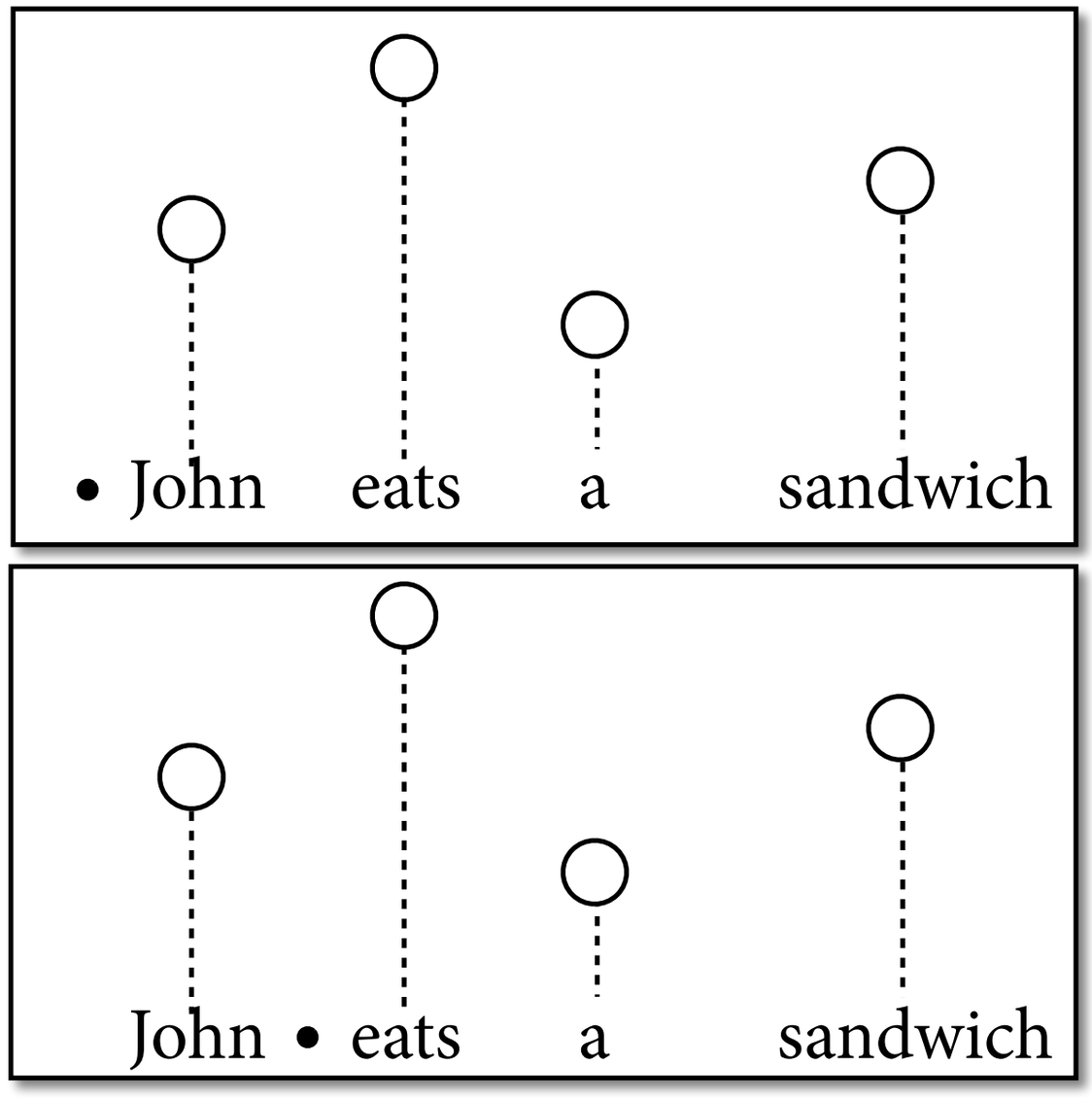
John eats a sandwich

# Example run

(ε, J eats a sw)

⇓ Shift

(J, eats a sw)




John eats a sandwich

# Example run

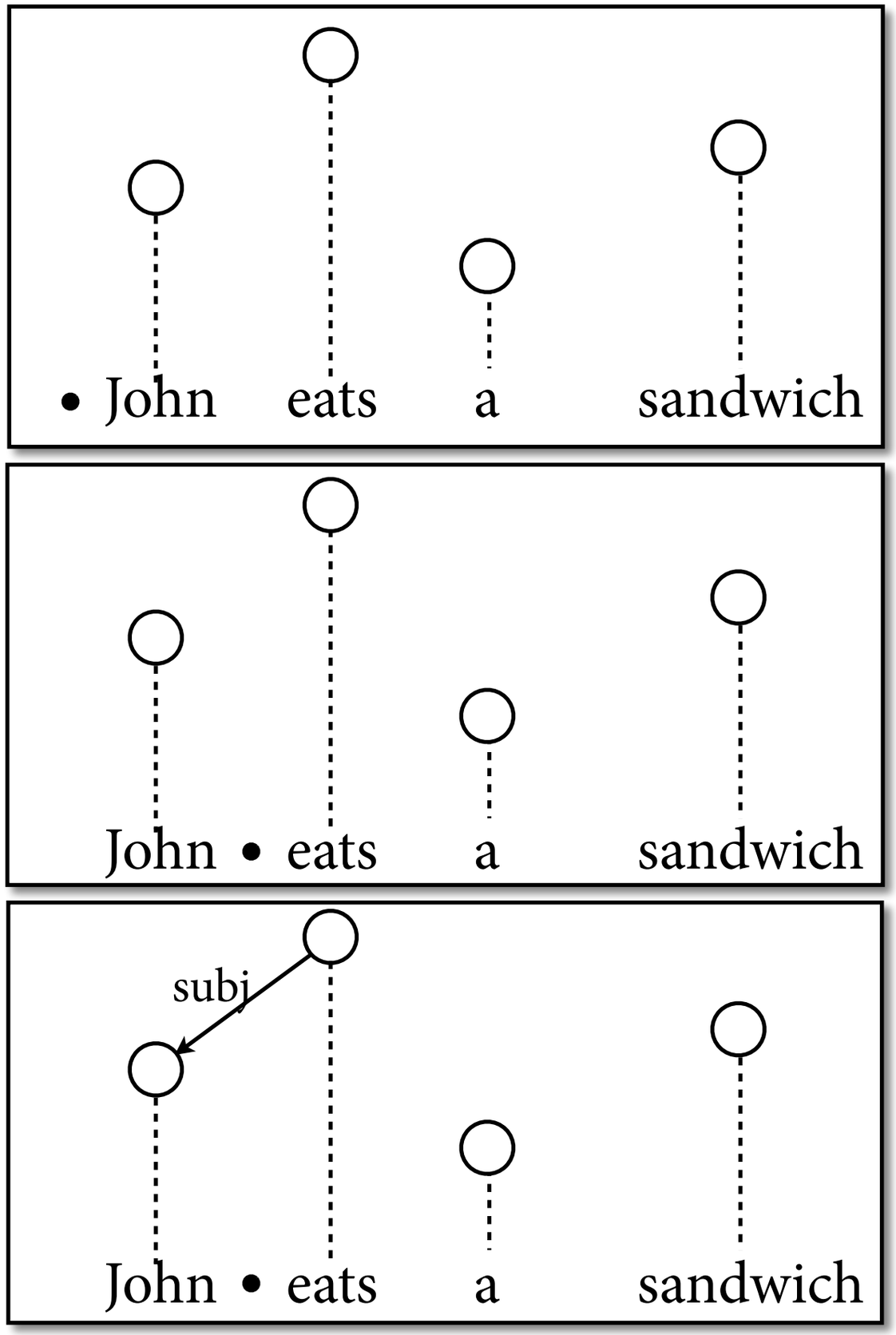
(ε, J eats a sw)

⇓ Shift

(J, eats a sw)

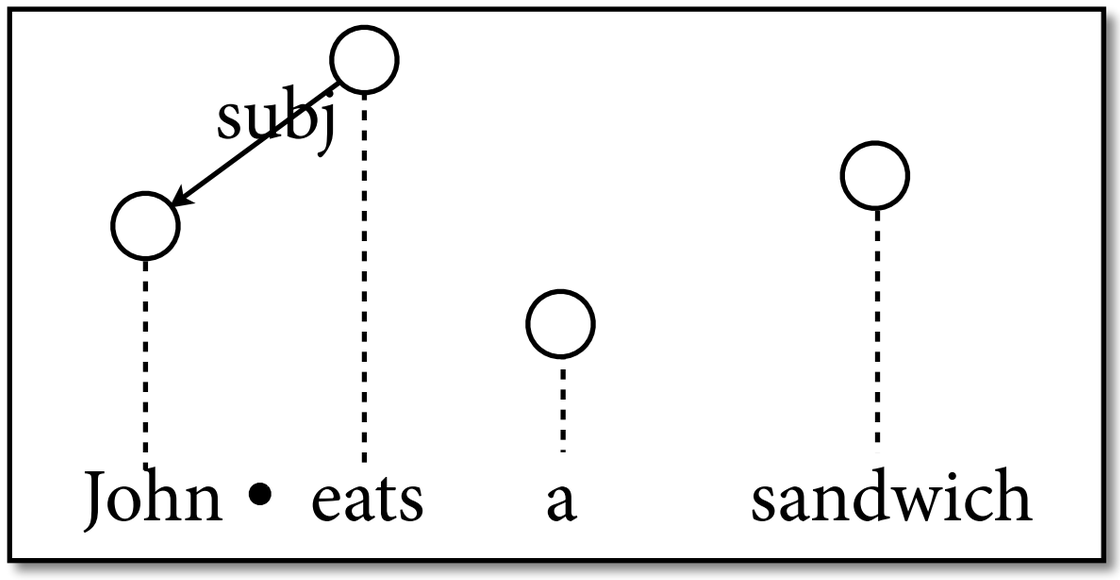
⇓ Left-Arc(subj)

(ε, eats a sw)



# Example run

(ε, eats a sw)

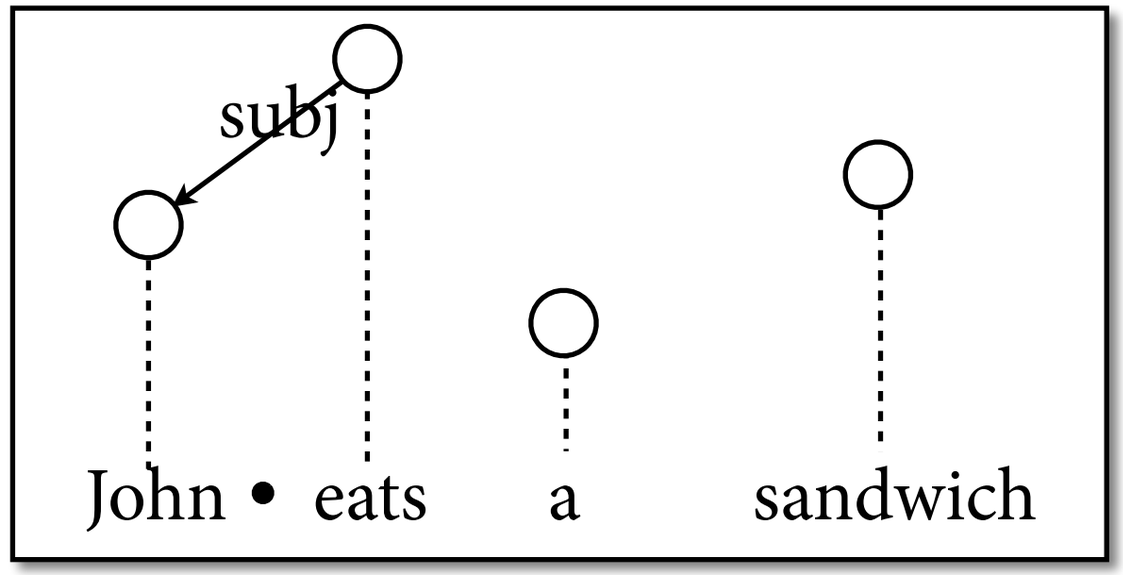



John eats a sandwich

John eats a sandwich

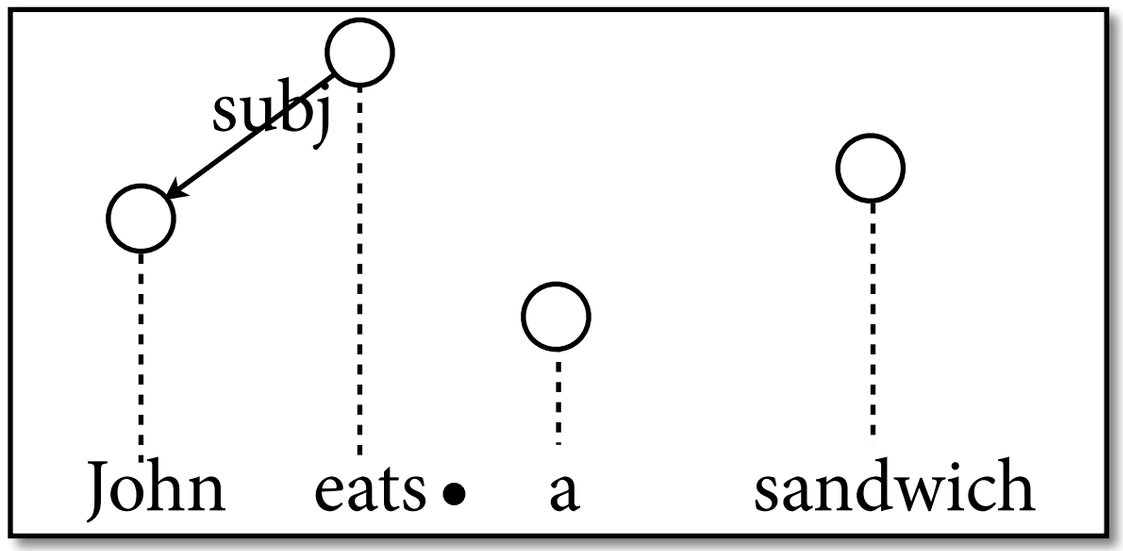
# Example run

(ε, eats a sw)




⇓ Shift

(eats, a sw)

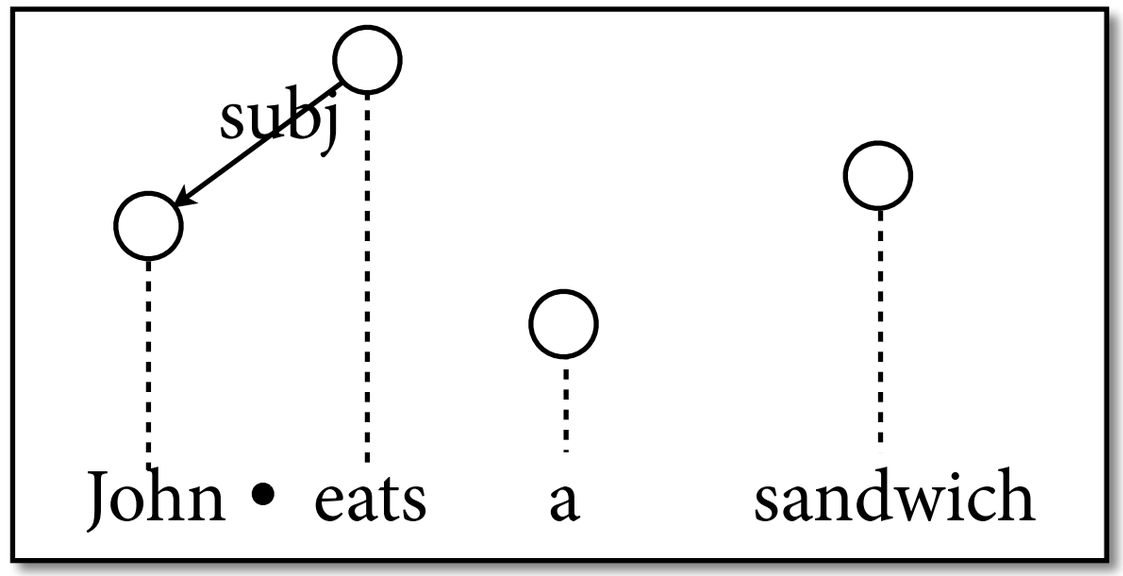



John eats a sandwich

# Example run

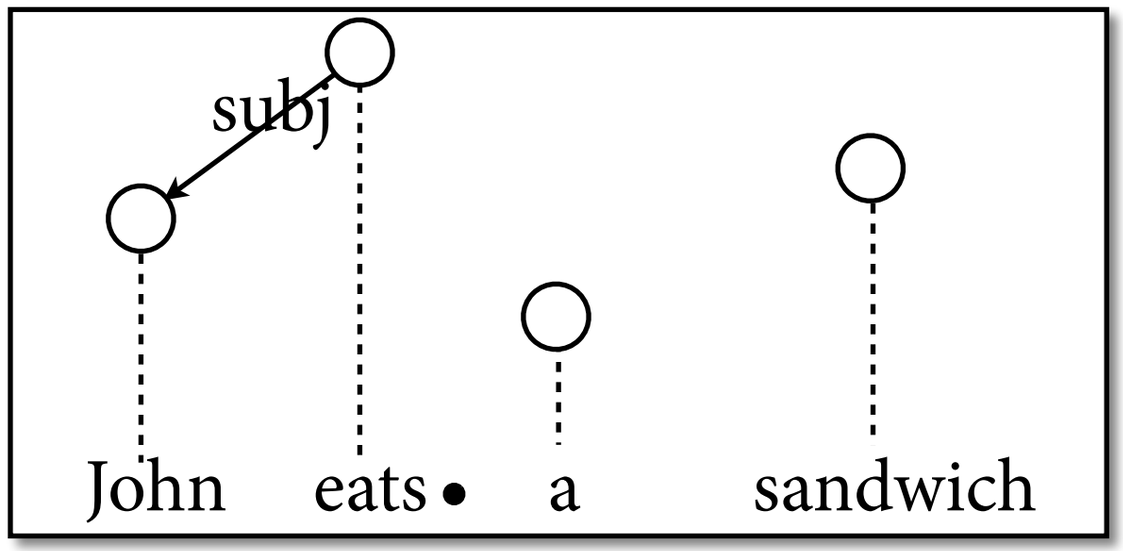
(ε, eats a sw)

⇓ Shift

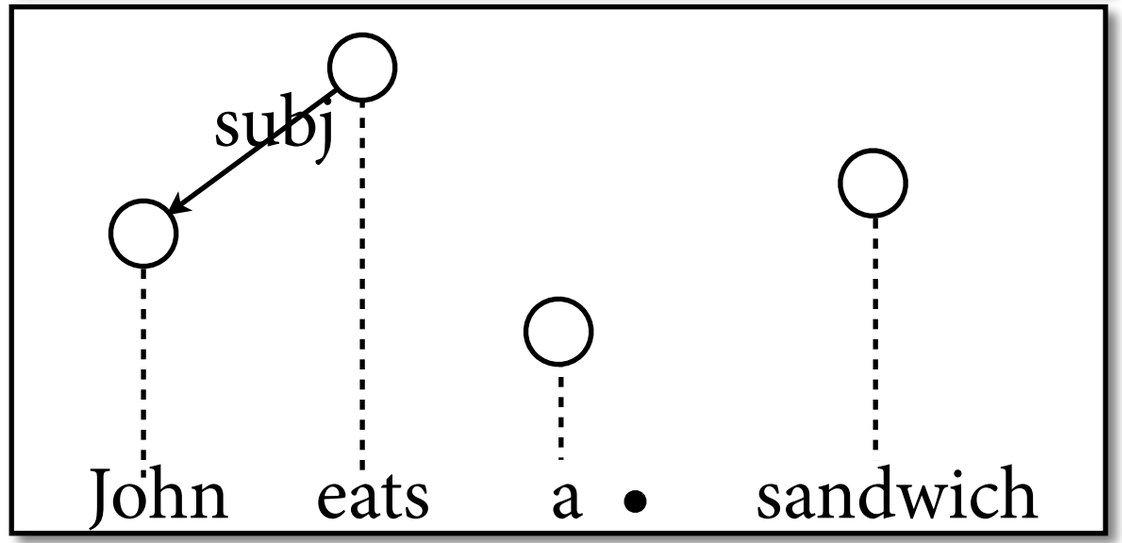



(eats, a sw)

⇓ Shift

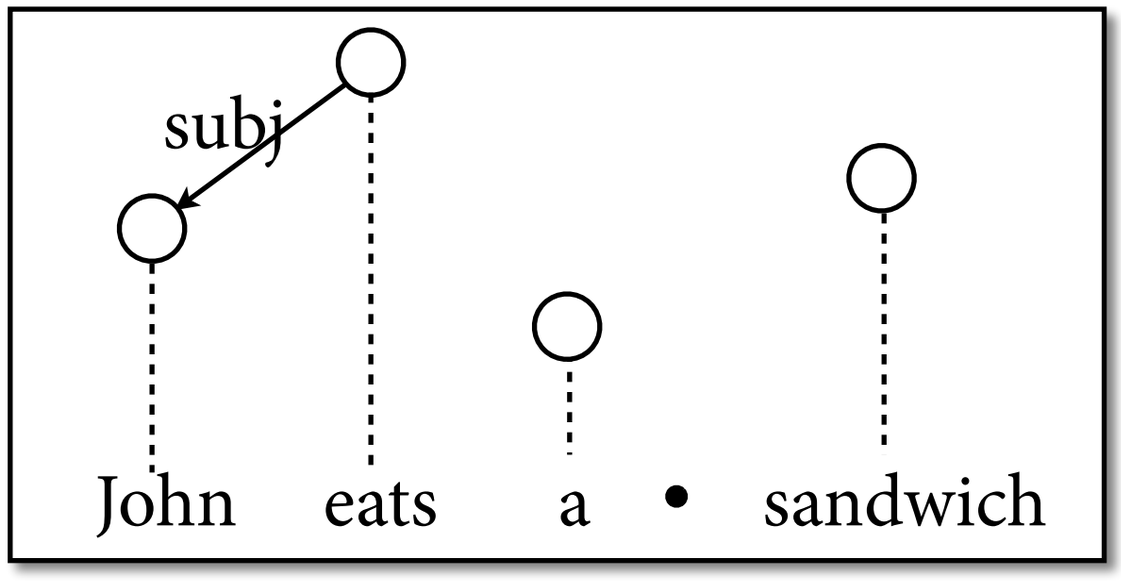



(eats a, sw)



# Example run

(eats a, sw)

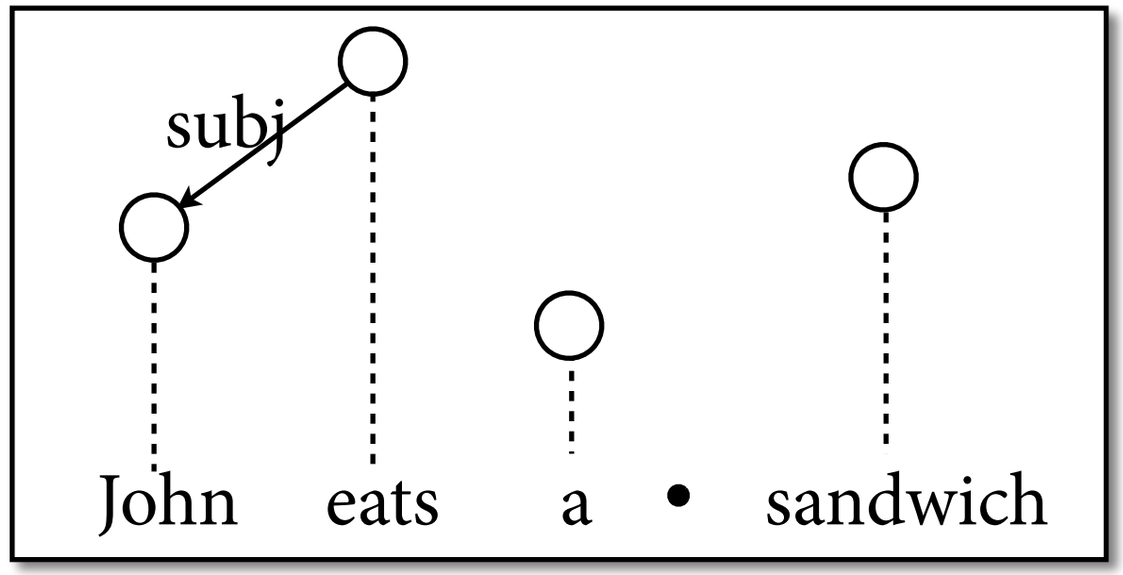



John eats a • sandwich

John eats a sandwich

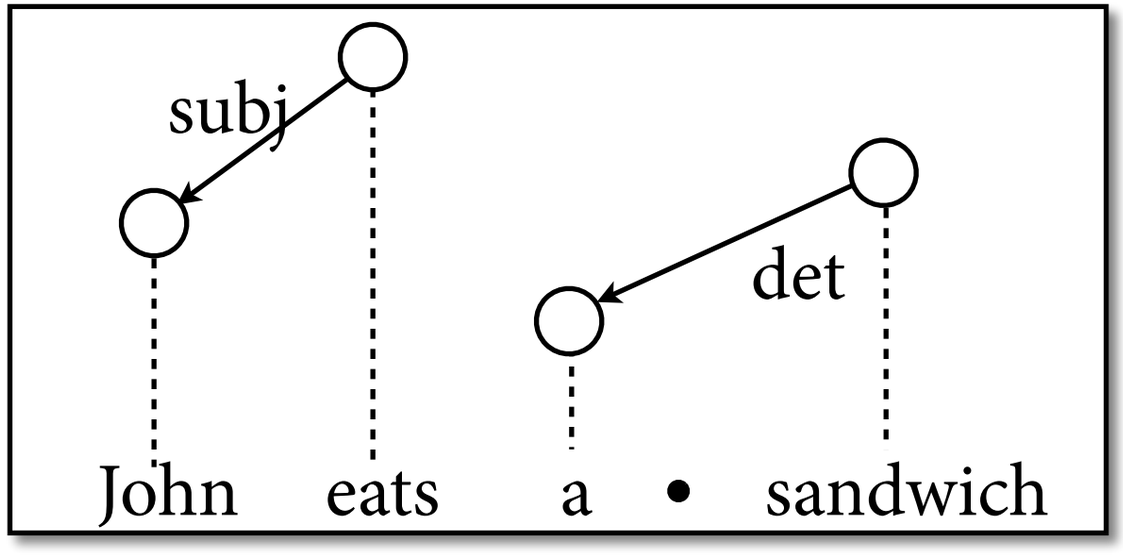
# Example run

(eats a, sw)




⇓ Left-Arc(det)

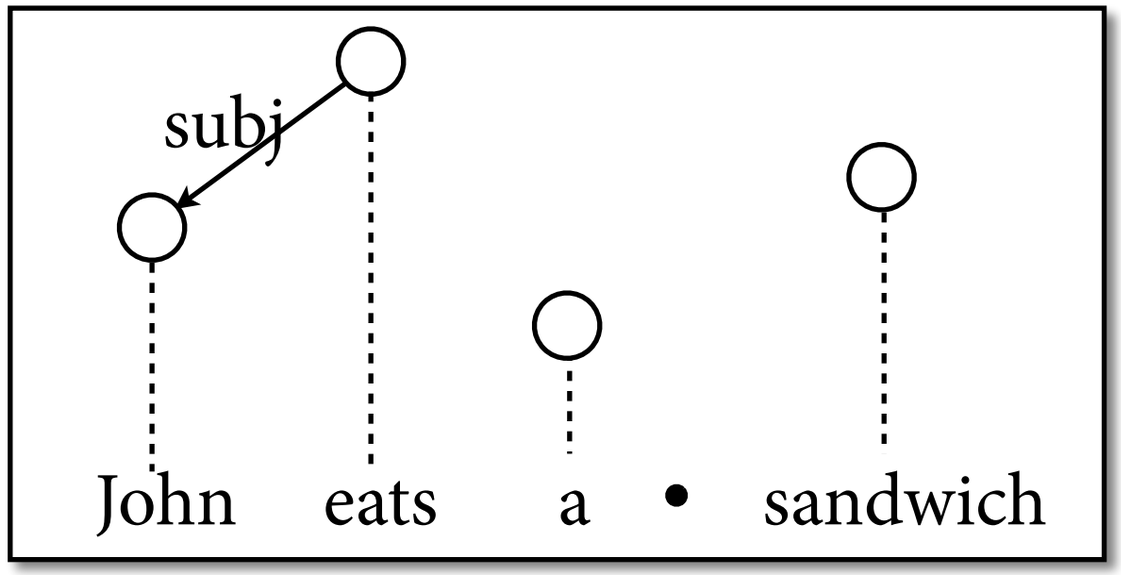
(eats, sw)




John eats a sandwich

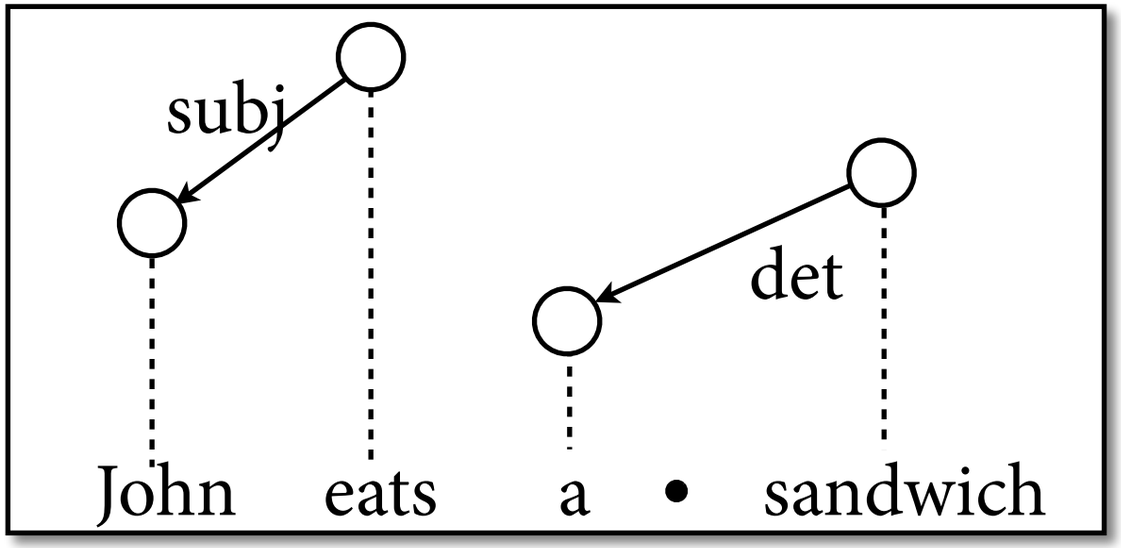
# Example run

(eats a, sw)



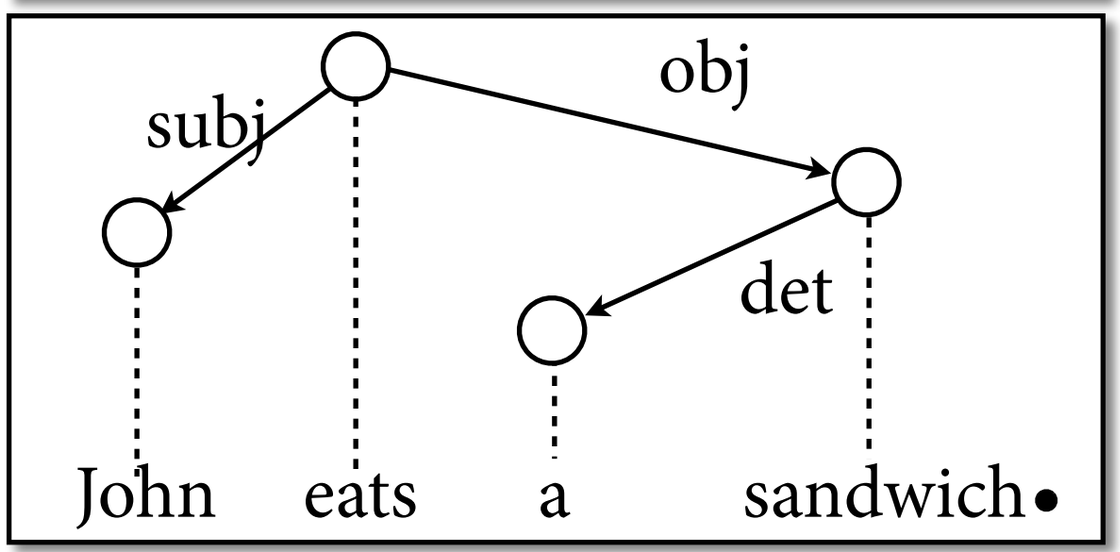

⇓ Left-Arc(det)

(eats, sw)




⇓ Right-Arc(obj)

(eats sw, ε)



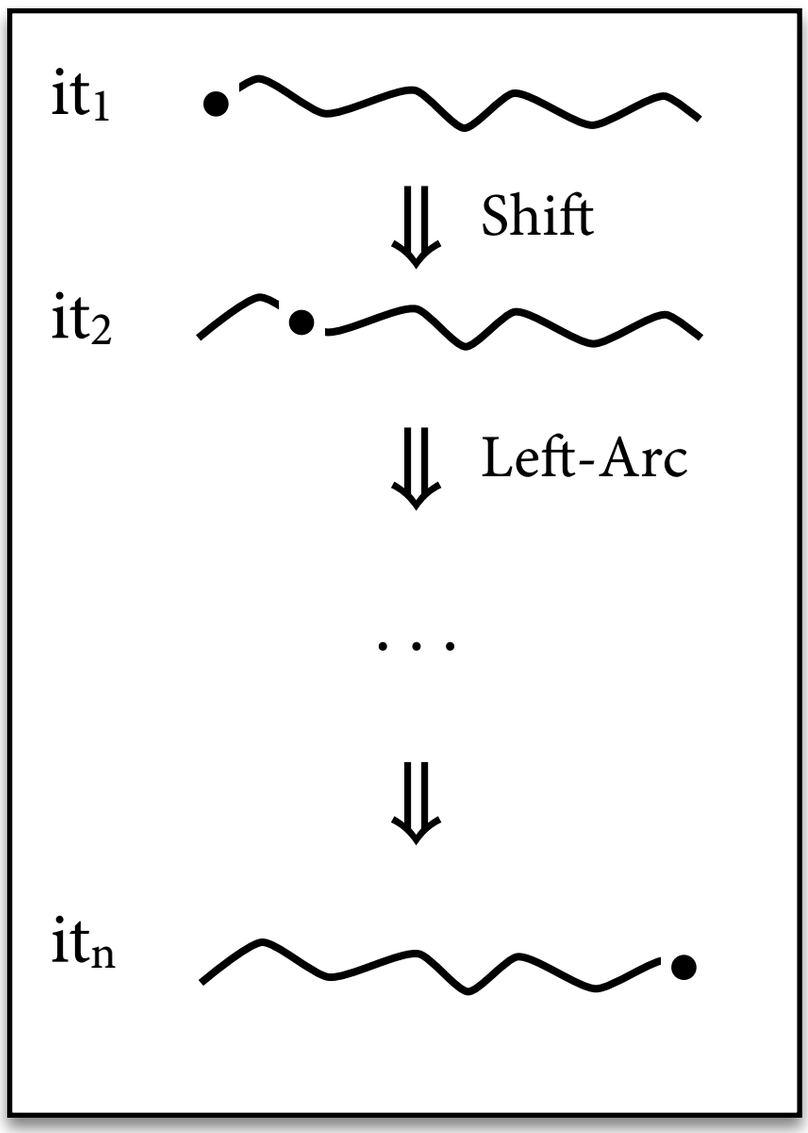
# Parsing as Classification

- Can now do deterministic parsing as follows:

```
c = start-item
while (c not goal-item and can apply
        at least one parsing operation to c):
    op = next-operation(c)
    c = perform-operation(c, op)
```

- "next-operation" chooses parsing operation to be applied to c. How do we get it?

# Learning classifier




Corpus of derivations

$(ft_1(it_1), ..., ft_m(it_1), \text{Shift})$

$(ft_1(it_2), ..., ft_m(it_2), \text{Left-Arc})$

. . .

$(ft_1(it_{n+1}), ..., ft_m(it_{n+1}), \text{Shift})$

Feature vectors

Train classifier

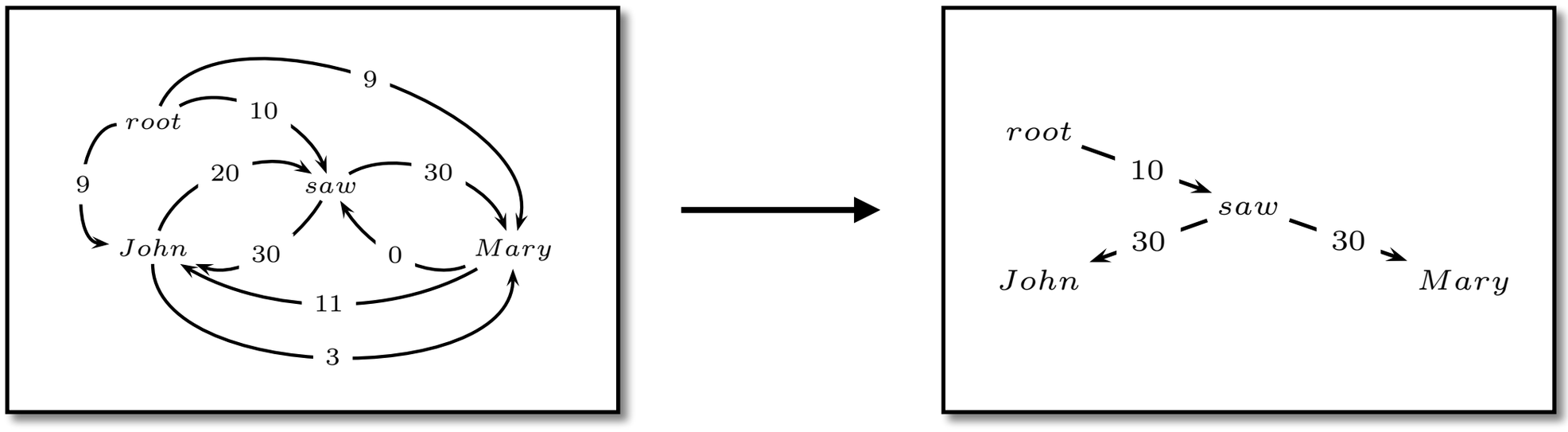
# Features in MaltParser

- MaltParser (= standard implementation of Nivre algorithm) offers "toolbox" for features:
  - $\sigma_i$: i-th stack token (from the top)
  - $\tau_i$: i-th token in remaining input
  - h(x): parent of x in the tree
  - l(x), r(x): leftmost (rightmost) child of x in the tree
  - p(x): POS tag of x
  - d(x): edge label from h(x) into x
  - build arbitrary terms from these, e.g. $p(l(\sigma_0))$

- Instead of engineering the features, can also use neural network classifier → Google SyntaxNet.

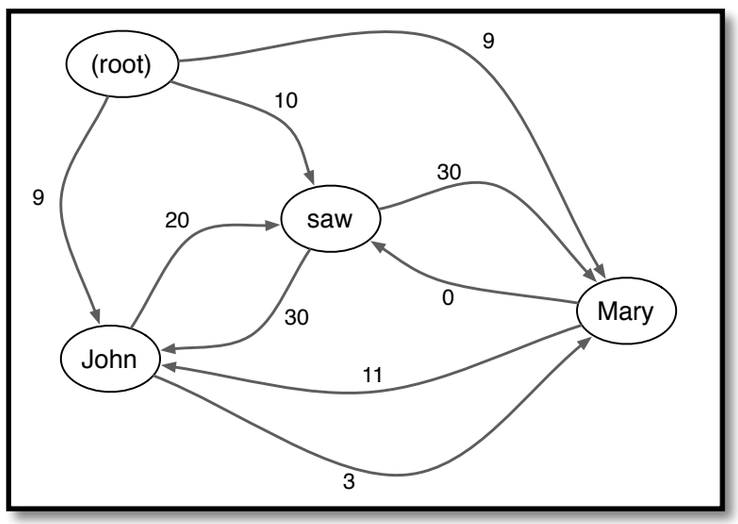
# The MST Parser

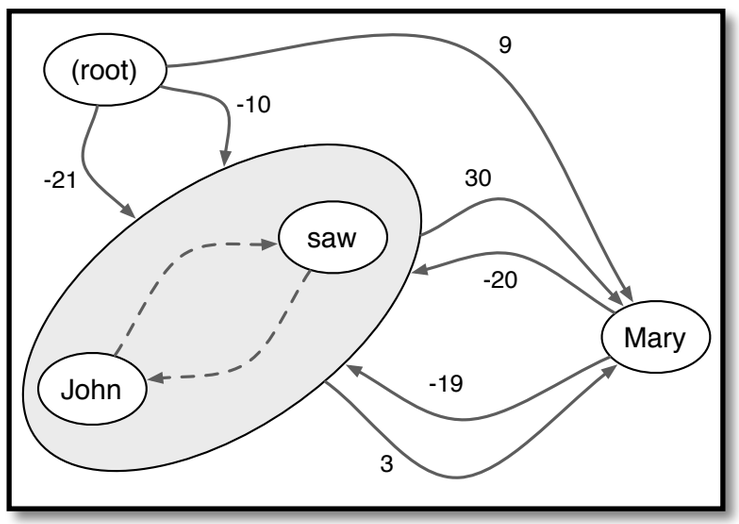
- Alternative idea (McDonald & Pereira, ca 2005):
  - take graph where nodes are words of sentence,
    and a directed edge between each two nodes
  - weight of edge represents how plausible a statistical model finds this edge
  - then calculate *maximum spanning tree*, i.e. tree that contains all nodes and has maximum sum of edge weights.



# Computing MSTs

Using the Chu-Liu-Edmonds algorithm, runtime $O(n^2)$



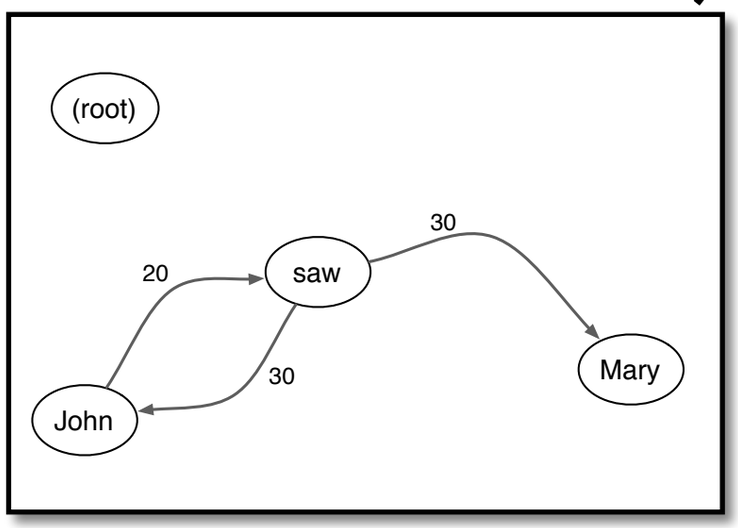



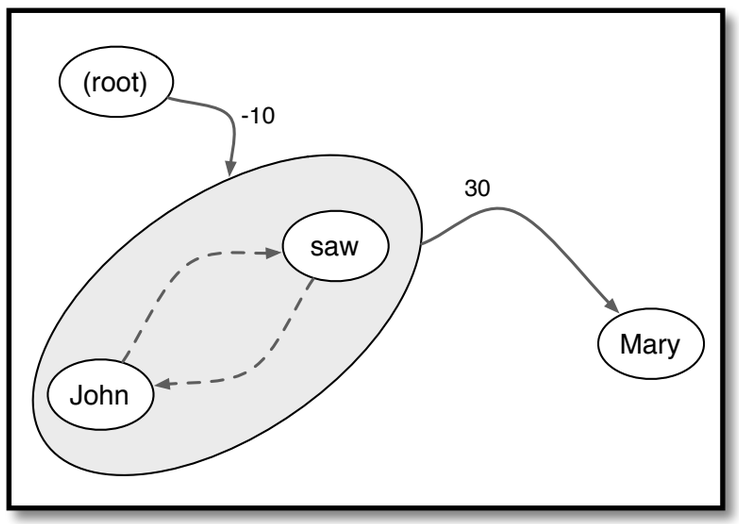

weight of new edge =
weight of old edge (u,v)
- weight of best edge into v

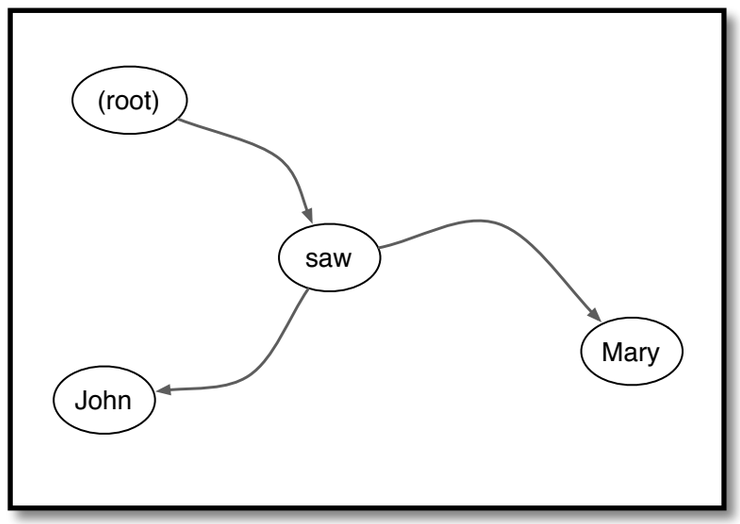
pick best incoming edges

contract cycles

pick best incoming edges



# Features

| Basic Uni-gram Features |
|---|
| p-word, p-pos |
| p-word |
| p-pos |
| c-word, c-pos |
| c-word |
| c-pos |

| Basic Big-ram Features |
|---|
| p-word, p-pos, c-word, c-pos |
| p-pos, c-word, c-pos |
| p-word, c-word, c-pos |
| p-word, p-pos, c-pos |
| p-word, p-pos, c-word |
| p-word, c-word |
| p-pos, c-pos |

| In Between POS Features |
|---|
| p-pos, b-pos, c-pos |
| **Surrounding Word POS Features** |
| p-pos, p-pos+1, c-pos-1, c-pos |
| p-pos-1, p-pos, c-pos-1, c-pos |
| p-pos, p-pos+1, c-pos, c-pos+1 |
| p-pos-1, p-pos, c-pos, c-pos+1 |

p = parent; c = child; b = word between parent and child in string

- Learn weight for each feature from training data.
  - using MIRA algorithm, which tries to maximize difference between score of correct parse and score of best wrong parse
- Instead of features, can also use neural model, e.g. Kiperwasser & Goldberg 2016.

# Evaluation

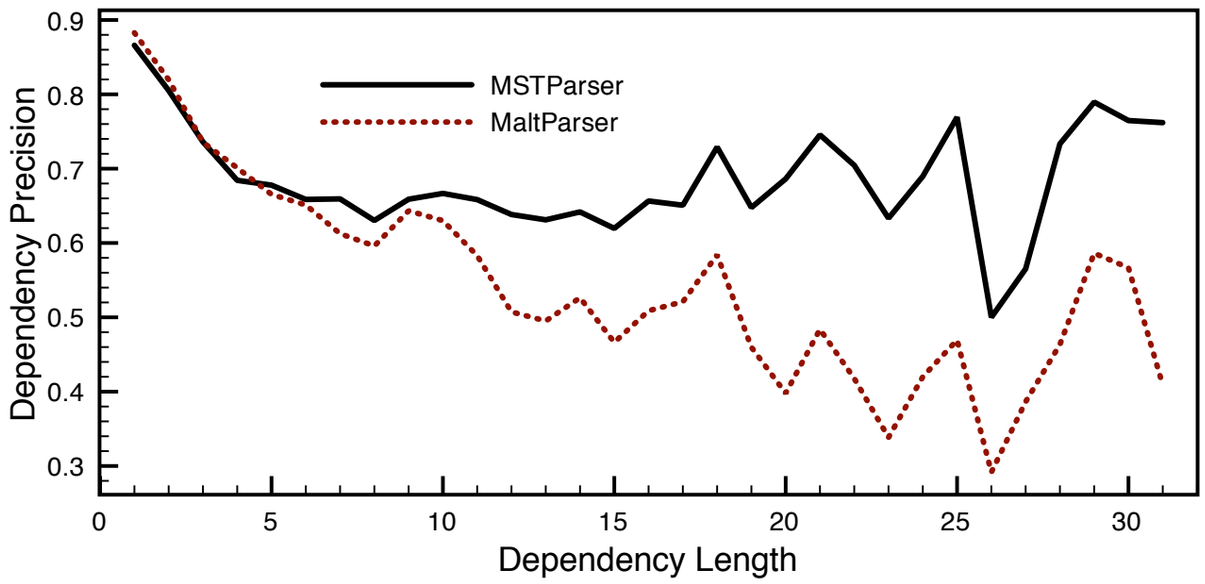
- Which proportion of edges predicted correctly?
  - *label accuracy:*
    #(nodes with correct label of incoming edge) / #nodes
  - *unlabeled attachment score:*
    #(nodes with correct parent) / #nodes
  - *labeled attachment score (LAS):*
    #(nodes with correct parent and edge label) / #nodes

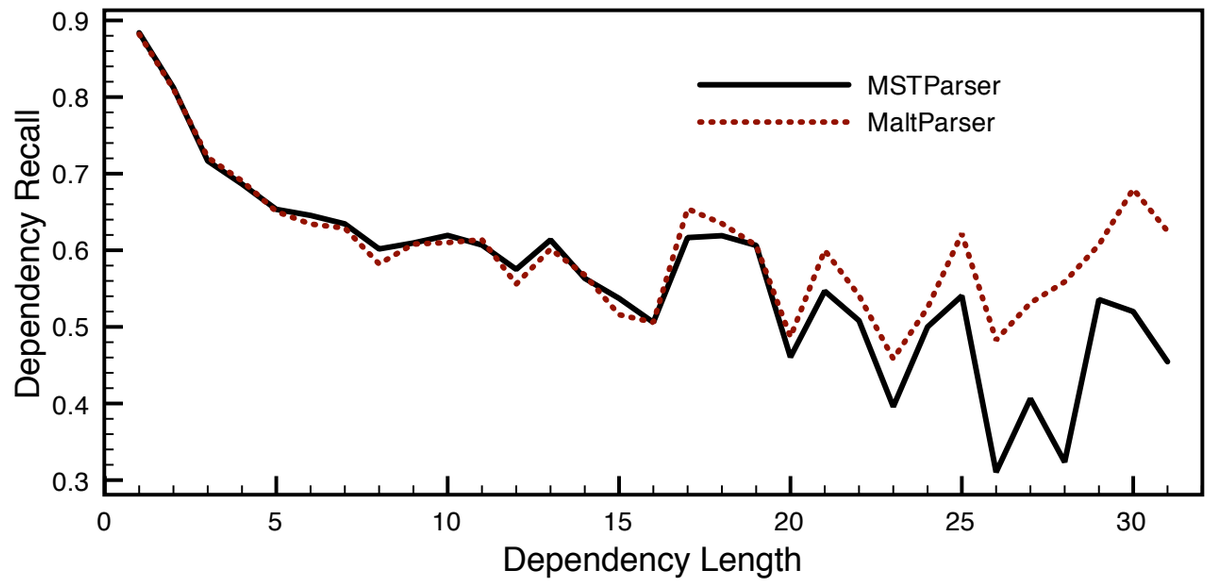
# Nivre vs McDonald

| | McDonald | Nivre |
|---|---|---|
| Arabic | 66.91 | 66.71 |
| Bulgarian | 87.57 | 87.41 |
| Chinese | 85.90 | 86.92 |
| Czech | 80.18 | 78.42 |
| Danish | 84.79 | 84.77 |
| Dutch | 79.19 | 78.59 |
| German | 87.34 | 85.82 |
| Japanese | 90.71 | 91.65 |
| Portuguese | 86.82 | 87.60 |
| Slovene | 73.44 | 70.30 |
| Spanish | 82.25 | 81.29 |
| Swedish | 82.55 | 84.58 |
| Turkish | 63.19 | 65.68 |
| Overall | 80.83 | 80.75 |

LAS in CoNLL-X Shared Task on Multilingual Dependency Parsing (2006)

# Comparison




- Observation (McDonald & Nivre 07): MaltParser and MSTParser make complementary mistakes.
  - MSTParser computes globally optimal tree, whereas MaltParser predicts local parsing choices.
  - MSTParser features can only look at individual edges, whereas MaltParser features can look at global tree structure.

# Summary

- Dependency parsing: fundamentally different style of parsing algorithm than with PCFGs.

- Much newer parsing style, but now just as popular as PCFG parsing in current research.

- Very fast in practice (e.g. MaltParser is O(n)); Google SyntaxNet does ~600 words/sec.

- State of the art:
  - LAS around 92 on English, around 90 on German
  - cool recent work trains on one language, directly used to parse a different one (with Universal Dependencies)