# Computational Linguistics

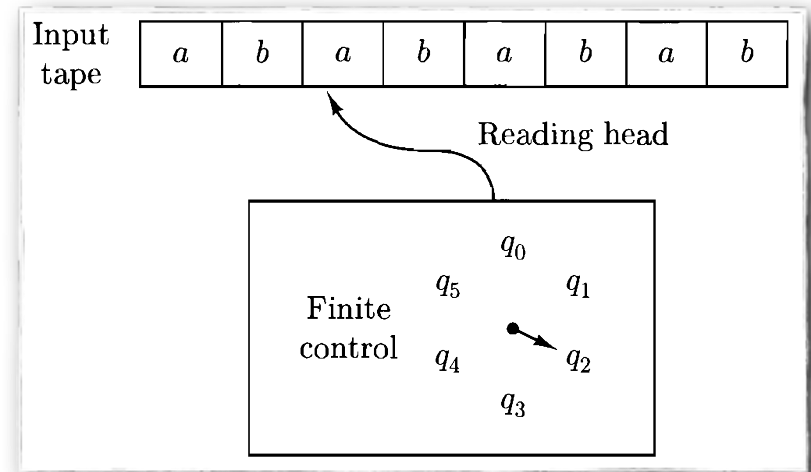
## Lecture 2 – Finite State Automata

Dietrich Klakow & Stefan Thater
FR 4.7 Allgemeine Linguistik (Computerlinguistik)
Universität des Saarlandes

Summer 2014

# Deterministic Finite Automata

- **M = ⟨K, Σ, δ, s, F⟩**
  - K  is a finite set of states
  - Σ  is an input alphabet
  - δ  is a transition function
  - s  ∈ K is the start state
  - F  ⊆ K is the set of final (accepting) states




- **Transition function**
  - δ(q, a) = q'
  - when M is in state q and reads input a, it goes into state q'

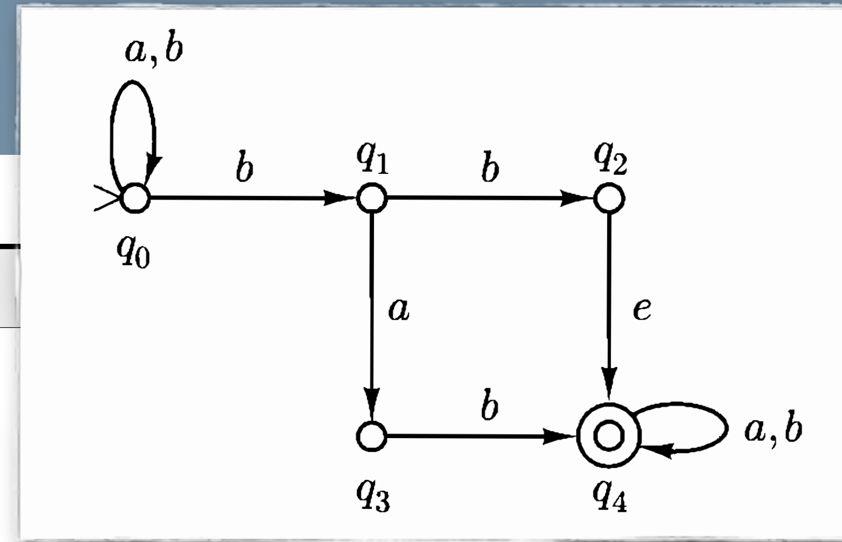
# Nondeterministic Automata

- **$M = \langle K, \Sigma, \Delta, s, F \rangle$**
  - K is a finite set of states
  - $\Sigma$ is an input alphabet
  - $\Delta \subseteq K \times \Sigma^* \times K$ is a finite transition relation
  - s $\in K$ is the start state
  - F $\subseteq K$ is the set of final (accepting) states
- **Transition relation** $\Delta \subseteq K \times \Sigma^* \times K$
  - $\langle q, w, q' \rangle \in \Delta$ = when the automaton is in state q and reads input w, it can go into state q'
  - Note: here we restrict ourself to NFA where $|w| \leq 1$

# Recognition Algorithm

```
function RECOGNIZE(NFA M, STRING input)
  agenda = list of configurations, initially containing only
           the configuration (start state of M, input)
  while agenda is not empty do
    conf ← POP(agenda)
    if conf is an accepting configuration then
      return accept
    else
      for all conf’ such that conf ⊢ conf’ do
        PUSH(agenda, conf’)
      end
  end
  return reject
end
```

# An Example




| conf | agenda |
|---|---|
| – | ⟨$q_0$, babba⟩ |
| ⟨$q_0$, babba⟩ | **⟨$q_0$, abba⟩, ⟨$q_1$, abba⟩** |
| ⟨$q_0$, abba⟩ | **⟨$q_0$, bba⟩**, ⟨$q_1$, abba⟩ |
| ⟨$q_0$, bba⟩ | **⟨$q_0$, ba⟩, ⟨$q_1$, ba⟩**, ⟨$q_1$, abba⟩ |
| ⟨$q_0$, ba⟩ | **⟨$q_0$, a⟩, ⟨$q_1$, a⟩**, ⟨$q_1$, ba⟩, ⟨$q_1$, abba⟩ |
| ⟨$q_0$, a⟩ | **⟨$q_0$, ε⟩**, ⟨$q_1$, a⟩, ⟨$q_1$, ba⟩, ⟨$q_1$, abba⟩ |
| ⟨$q_0$, ε⟩ | ⟨$q_1$, a⟩, ⟨$q_1$, ba⟩, ⟨$q_1$, abba⟩ |
| ⟨$q_1$, a⟩ | **⟨$q_3$, ε⟩**, ⟨$q_1$, ba⟩, ⟨$q_1$, abba⟩ |
| ⟨$q_3$, ε⟩ | ⟨$q_1$, ba⟩, ⟨$q_1$, abba⟩ |
| ⟨$q_1$, ba⟩ | **⟨$q_2$, a⟩**, ⟨$q_1$, abba⟩ |
| ⟨$q_2$, a⟩ | **⟨$q_4$, a⟩**, ⟨$q_1$, abba⟩ |
| ⟨$q_4$, a⟩ | **⟨$q_4$, ε⟩**, ⟨$q_1$, abba⟩ |
| ⟨$q_4$, ε⟩ | ⟨$q_1$, abba⟩ |

# NFA = DFA (preliminary)

- **Theorem:** for every NFA $M = \langle K, \Sigma, \Delta, s, F\rangle$ there is an equivalent DFA $M'$ such that $L(M) = L(M')$
- Let us first consider the special case where for all elements $\langle q, w, q'\rangle \in \Delta$, $w$ is a string of length 1
- We construct the DFA $M' = \langle K', \Sigma, \delta, s', F'\rangle$ as follows:
  - $K' = 2^K$
  - $s' = \{s\}$
  - $\delta(Q, a) = \{ k \in K \mid \langle q, a, k\rangle \in \Delta \text{ for some } q \in Q \}$
    - for all $Q \subseteq K$, $a \in \Sigma$
  - $F' = \{ Q \subseteq K \mid Q \cap F \neq \emptyset \}$

# Subset construction algorithm

```
function DFA(K, Σ, Δ, s, F)
  K’ ← list that contains only ε-closure(s), unmarked
  while there is an unmarked state T in K’ do
    mark T
    for each symbol a ∈ Σ do
      U ← ε-closure(move(T, a))
      if U ∉ K’ then
        add U as an unmarked state to K’
      δ[T, a] ← U
    end
  end
  return <the corresponding DFA>
end
```

# Computational Linguistics

## Lecture 3 – Parsing

Dietrich Klakow & Stefan Thater
FR 4.7 Allgemeine Linguistik (Computerlinguistik)
Universität des Saarlandes

Summer 2014

# Grammars

- Grammars generate sentences (“words”)

| | |
|---|---|
| S → NP VP | DET → *the* |
| NP → DET N | DET → *a* |
| NP → NP PP | N → *student* |
| PP → P NP | N → *book* |
| VP → V | N → *library* |
| VP → V NP | V → *works* |
| VP → VP PP | V → *reads* |
| | P → *in* |

*The student works*

*The student works in the library*

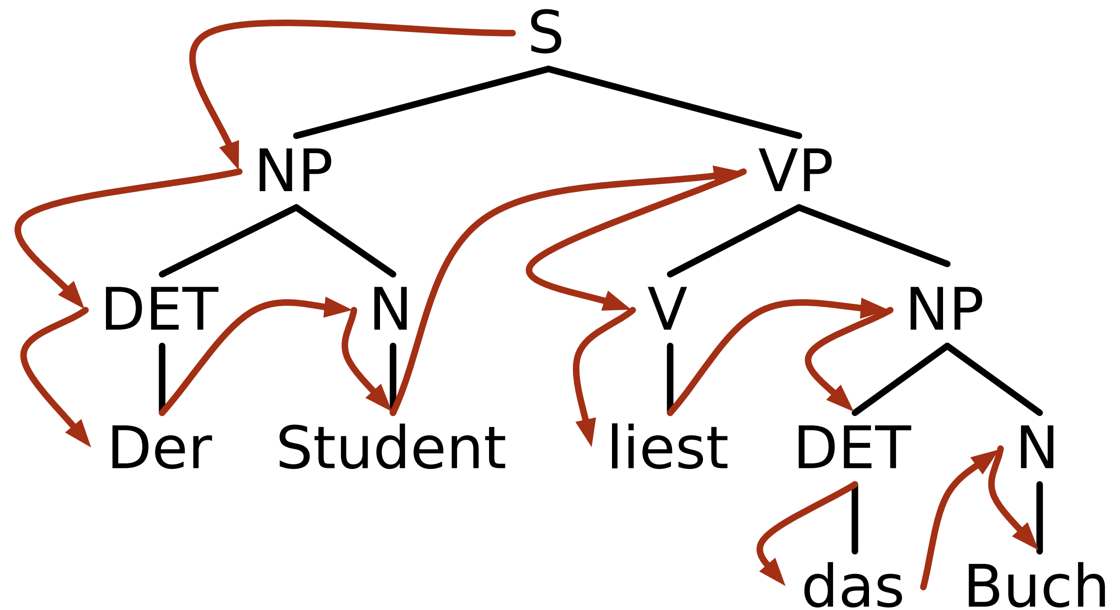
*The student reads a book*

*The student reads a book in the library*

*[...]*

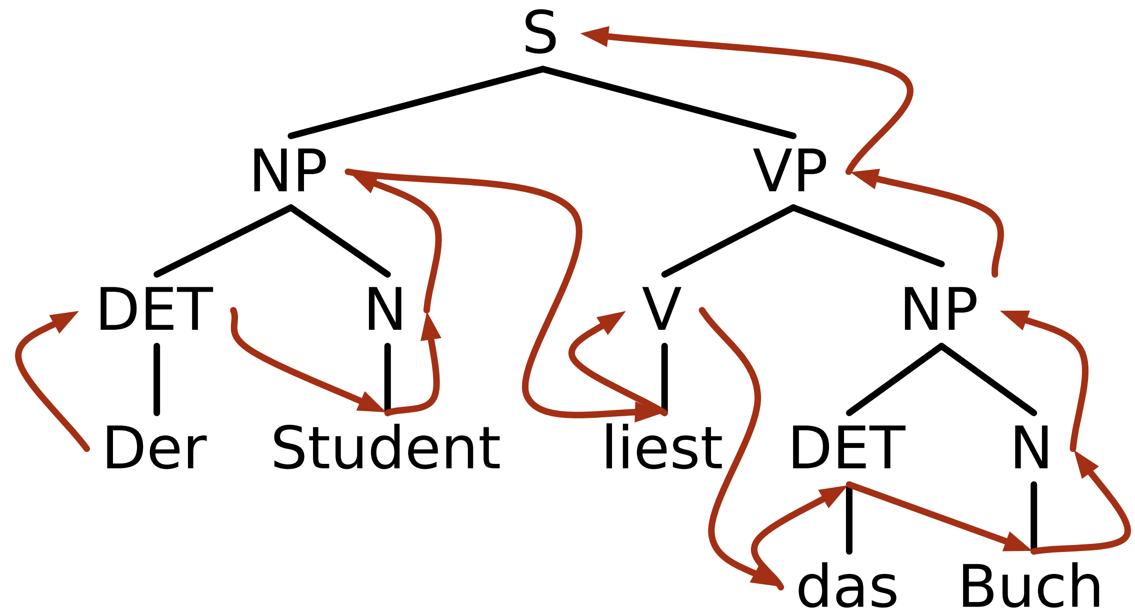
# Basic Parsing Strategies

- **A top-down parser / recognizer …**
  - starts with the start symbol (= root node)
  - applies production rules “from left to right”
  - and tries to match the input sequence



# Basic Parsing Strategies

- **A bottom-up parser / recognizer ...**
  - starts with the input sequence (= leaf nodes)
  - scans the input for subsequences that match the right-hand side of some rule and applies it “from right to left”



# Shift-Reduce Parsing (Bottom-up)

- **Initial configuration** for input sequence $w_1 \dots w_n$:
  - $\langle [], [w_1, \dots, w_n] \rangle$

stack

the input that still needs
to be processed

- **Accepting configuration**
  - $\langle [S], [] \rangle$
- In each step we can perform …
  - shift – move a symbol to the stack
  - reduce – apply a matching rule to the topmost elements on the stack

# Shift

- The shift operation moves one symbol to the stack
- **Configuration:**
  - $\langle [A_1, \ldots, A_k], [w_i, w_{i+1}, \ldots, w_n] \rangle$
- **New configuration:**
  - $\langle [A_1, \ldots, A_k, w_i], [w_{i+1}, \ldots, w_n] \rangle$

# Reduce

- Reduce replaces the topmost symbols on the stack by the lefthand side of a matching rule
- **Configuration:**
  - $\langle [A_1, \ldots, A_{j-1}, A_j, \ldots, A_k], [w_i, \ldots, w_n] \rangle$
- **Rule:**
  - $B \rightarrow A_j, \ldots, A_k$
- **New Configuration:**
  - $\langle [A_1, \ldots, A_{j-1}, B], [w_i, \ldots, w_n] \rangle$

# Example – *The student works*

| | ⟨stack, sent⟩ | agenda |
|---|---|---|
| 1 | – | ⟨[] [the student works]⟩ |
| 2 | ⟨[] [the student works]⟩ | ⟨[the] [student works]⟩ |
| 3 | ⟨[the] [student works]⟩ | ⟨[DET] [student works]⟩ ⟨[the student] [works]⟩ |
| 4 | ⟨[DET] [student works]⟩ | ⟨[DET student] [works]⟩ ⟨[the student] [works]⟩ |
| 5 | ⟨[DET student] [works]⟩ | ⟨[DET N] [works]⟩ ⟨[DET student works] []⟩ ⟨[the student] [works]⟩ |
| 6 | ⟨[DET N] [works]⟩ | ⟨[NP] [works]⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ … |
| 7 | ⟨[NP] [works]⟩ | ⟨[NP works] []⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ … |
| 8 | ⟨[NP works] []⟩ | ⟨[NP V] []⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ … |
| 9 | ⟨[NP V] []⟩ | ⟨[NP VP] []⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ … |
| 10 | ⟨[NP VP] []⟩ | ⟨[S] []⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ … |
| 11 | ⟨[S] []⟩ | ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ … |

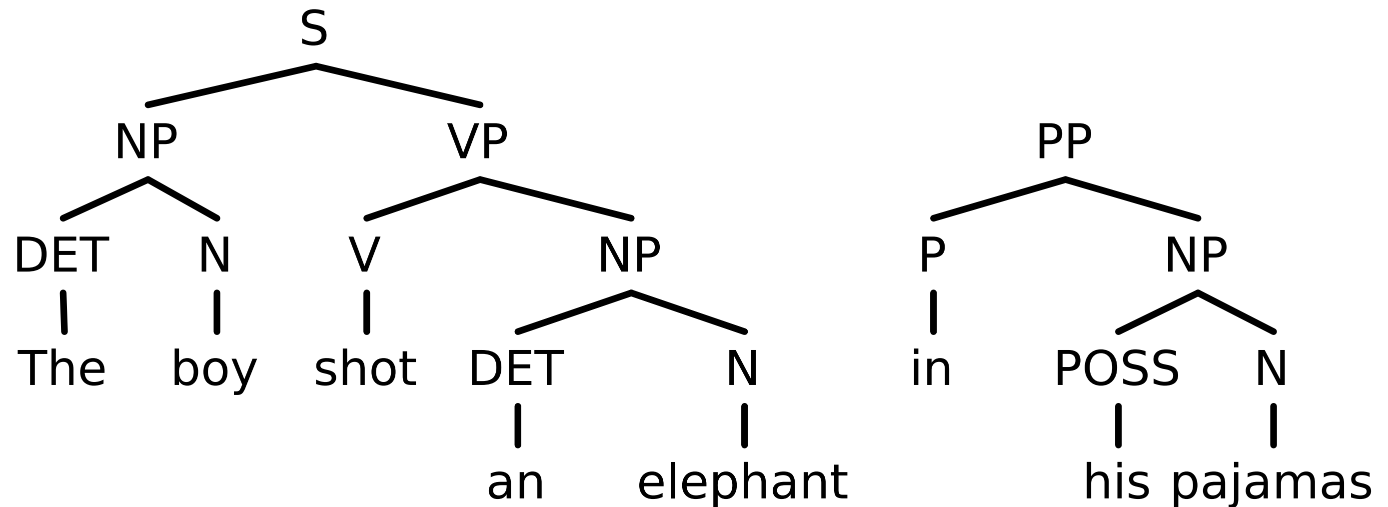
# *The boy shot an elephant in ...*

⟨[], [*the boy shot an elephant in his pajamas*]⟩

$\Rightarrow^*$ ⟨[NP VP], [*in his pajamas*]⟩

$\Rightarrow$ ⟨[S], [*in his pajamas*]⟩

$\Rightarrow^*$ ⟨[S PP], []⟩ $\Rightarrow$ Failure, Backtracking



# Charts as Matrices

- **A ∈ T[i, j]** iff $A \Rightarrow^* w_{i+1} \ldots w_j$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | DET | | | | | | | |
| 2 | NP | N | | | | | | |
| 3 | ∅ | ∅ | V | | | | | |
| 4 | ∅ | ∅ | ∅ | DET | | | | |
| 5 | S | ∅ | VP | NP | N | | | |
| 6 | ∅ | ∅ | ∅ | ∅ | ∅ | P | | |
| 7 | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | POSS | |
| 8 | S | ∅ | VP | NP | ∅ | PP | NP | N |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | *The* | boy | *shot* | *an* | *elephant* | *in* | *his* | *pajamas* |

0 1 2 3 4 5 6 7 8

| | |
|---|---|
| S → NP VP | DET → *the* |
| NP → DET N | DET → *an* |
| NP → POSS N | N → *boy* |
| NP → NP PP | N → *elephant* |
| PP → P NP | N → *pajamas* |
| VP → V NP | V → *shot* |
| VP → VP PP | P → *in* |
| | POSS → *his* |

# CYK (Recognizer, Pseudo-code)

```
function CYK(G, w1 ... wn):
    for i in 1 ... n do
        T[i-1, i] = { A | A → wi ∈ R }
        for j in i - 2 ... 0 do
            T[j, i] = ∅
            for k in j + 1 ... i - 1 do
                T[j, i] = T[j, i] ∪
                    { A | A → B C, B ∈ T[j,k], C ∈ T[k, i] }
            done
        done
    done
    if S ∈ T[0, n] then return True else return False
```

# Binarization

**left binarization(G):**

while G contains rules $A \rightarrow A_1\ A_2\ A_3\ \ldots\ A_k$, $k \geq 3$

delete the rule from G

add rule $\langle A_1, \ldots, A_{k-1} \rangle \rightarrow A_1\ \ldots\ A_{k-1}$

add rule $A \rightarrow \langle A_1, \ldots, A_{k-1} \rangle\ A_k$

**right binarization(G):**

while G contains rules $A \rightarrow A_1\ A_2\ A_3\ \ldots\ A_k$, $k \geq 3$

delete the rule from G

add rule $\langle A_2, \ldots, A_k \rangle \rightarrow A_2\ \ldots\ A_k$

add rule $A \rightarrow A_1\ \langle A_2, \ldots, A_k \rangle$

# Implementation variants

- $T[i,j] = T[i,j] \cup \{ A \mid A \rightarrow B\ C, B \in T[i,k], C \in T[k,j] \}$
  - ⇒ can be implemented in different ways
- **Method 1**
  - Iterate over all rules $A \rightarrow B\ C$
  - Check if $B \in T[i,k]$ and $C \in T[k,j]$
- **Method 2**
  - Iterate over all $B \in T[i,k]$
  - Iterate over all rules $A \rightarrow B\ C$
  - Check if $C \in T[k, j]$

# Implementierungsvarianten

- $T[i,j] = T[i,j] \cup \{ A \mid A \rightarrow B\ C,\ B \in T[i,k],\ C \in T[k,j] \}$
  - ⇒ can be implemented in different ways
- **Method 3**
  - Iterate over all $C \in T[k,j]$
  - Iterate over all rules $A \rightarrow B\ C$
  - Check if $A \in T[i,k]$
- **Method 4**
  - Iterate over all $B \in T[i,k]$ and $C \in T[k,j]$
  - Check if a rule $A \rightarrow B\ C$ exists

# Computational Linguistics

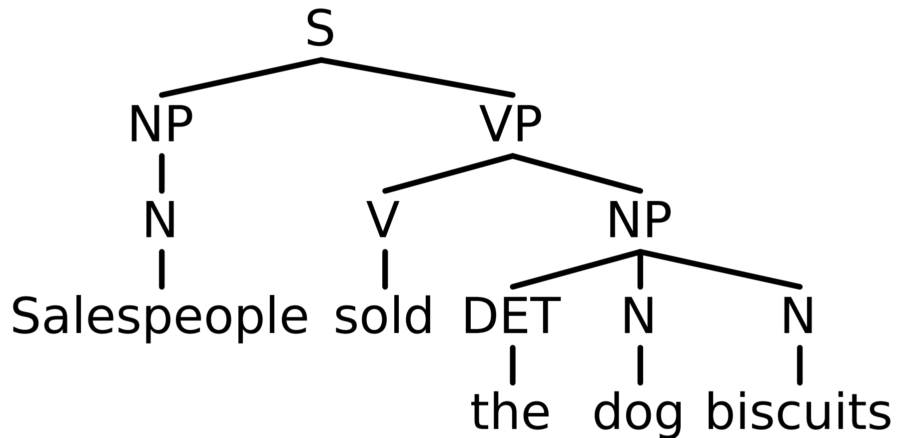
## Probabilistic Parsing

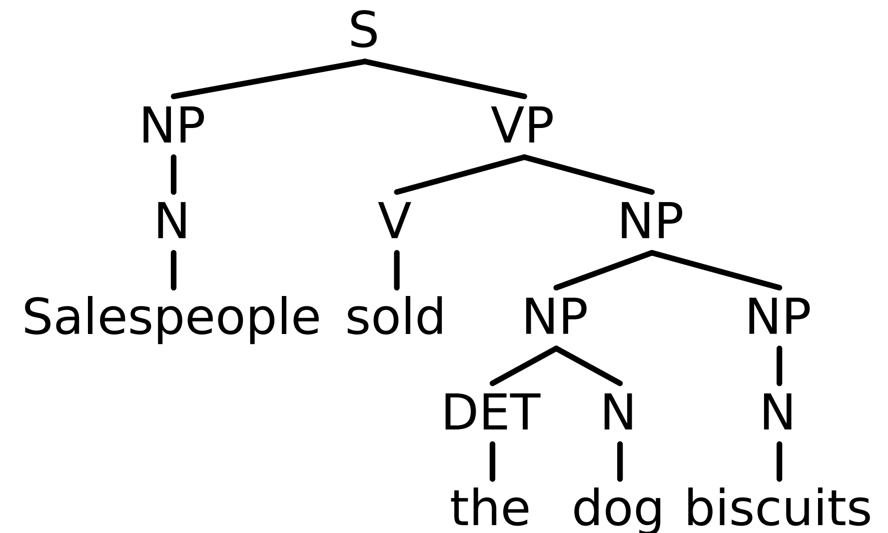
Dietrich Klakow & Stefan Thater
FR 4.7 Allgemeine Linguistik (Computerlinguistik)
Universität des Saarlandes

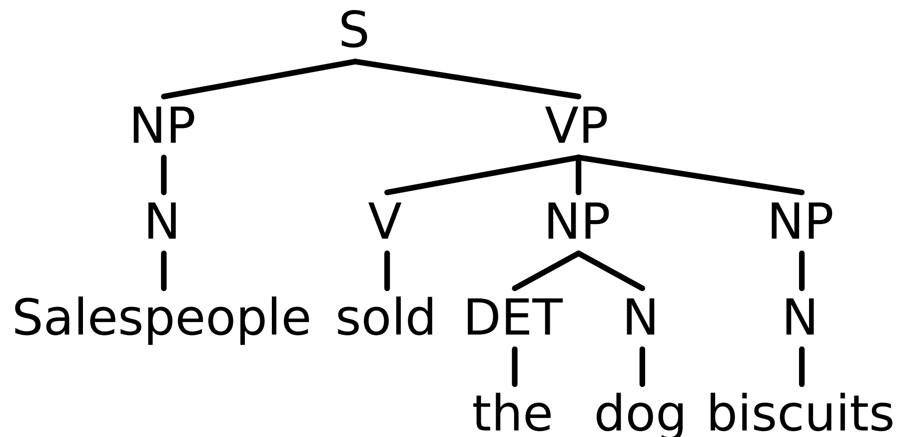
Summer 2014

(Charniak, 1997)

# *Salespeople sold the dog biscuits*




| | |
|---|---|
| S → NP VP | NP → NP NP |
| VP → V NP | NP → N |
| VP → V NP NP | DET → the |
| NP → DET N | N → dog |
| NP → DET N N | … |

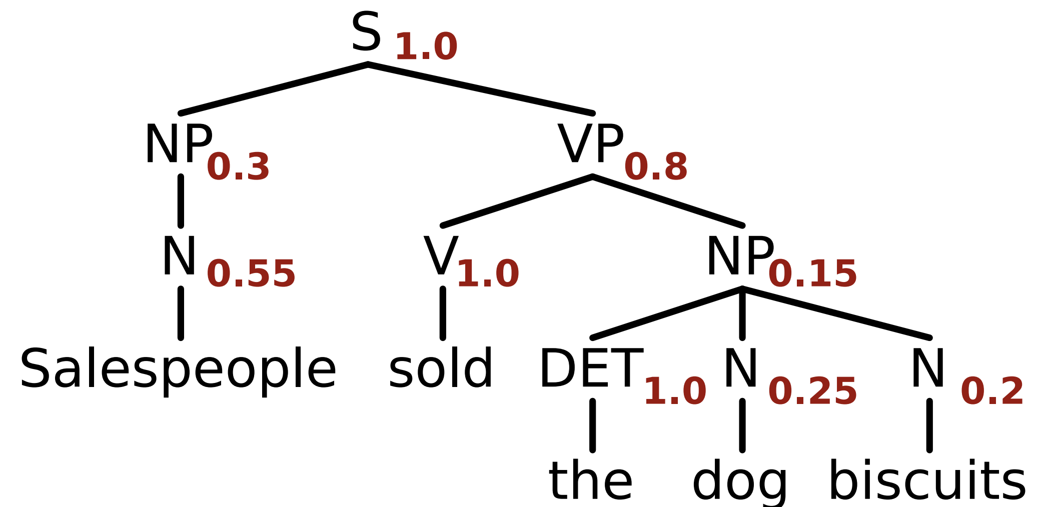
# Probabilistic Context-Free Grammars (PCFG)

- **Probabilistic context-free grammar (PCFG)**
  - a context-free grammar $\langle V, \Sigma, R, S \rangle$
  - a funktion P assigning a value $p \in [0, 1]$ to each rule
    - such that $\sum_{\beta \in V^*} P(A \rightarrow \beta) = 1$
- $P(A \rightarrow \beta)$ = the conditional probability that symbol A is expanded to β
  - Alternative notations: $P(\beta \mid A)$, $P(A \rightarrow \beta \mid A)$, $A \rightarrow \beta$ [p]

(Charniak, 1997)

# *Salespeople sold the dog biscuits*

| | | |
|---|---|---|
| S | → NP VP | [1.0] |
| VP | → V NP | [0.8] |
| VP | → V NP NP | [0.2] |
| NP | → DET N | [0.5] |
| NP | → N | [0.3] |
| NP | → DET N N | [0.15] |
| NP | → NP NP | [0.05] |
| DET | → *the* | [1.0] |
| N | → *Salespeople* | [0.55] |
| N | → *dog* | [0.25] |
| N | → *biscuits* | [0.2] |
| V | → *sold* | [1.0] |




$$P(t) = 1.0 \times 0.3 \times 0.55 \times 0.8 \times 1.0 \times 0.15 \times 1.0 \times 0.25 \times 0.2 = 9.9 \times 10^{-4}$$

# CYK (with probabilities)

```
function CYK(G, w₁ ... wₙ):
    ⟨initialize T and B⟩
    for i in 1 ... n do
        for all nonterminals A in G do
            T[i-1, i, A] = P(A → wᵢ)
        for j in i - 2 ... 0 do
            for k in j + 1 ... i - 1 do
                for all A → B C do
                    pr = T[j, k, B] × T[k, i, C] × P(A → B C)
                    if pr > T[j, i, A] then
                        T[j, i, A] = pr
                        B[j, i, A] = ⟨construct subtree⟩
    return ⟨B[0, n, S] and T[0, n, S]⟩
```

# Learning PCFG Probabilities

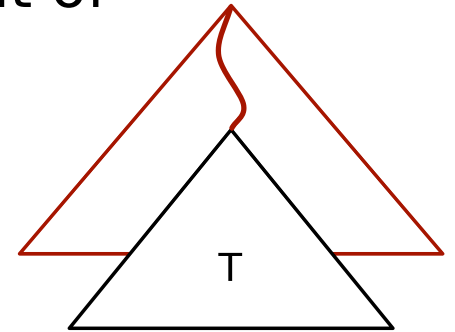
- We are given a syntactically annotated corpus
  - annotated corpus = a set of derivation trees
- We can construct a grammar from the treebank by identifying the rules with all “subtrees” of height 1
- **Estimating rule probabilities:**
  - $P(A \rightarrow \alpha) = \dfrac{\text{count}(A \rightarrow \alpha)}{\sum_{\beta} \text{count}(A \rightarrow \beta)}$
  - $\text{count}(A \rightarrow \alpha)$ = the number of times the rule $A \rightarrow \alpha$ has been used in all trees in the corpus

# Evaluation

- **Coverage:** How many sentences are well-formed according to the grammar?
- **Accuracy:** How many sentences are correctly parsed?
  - measured as “relative correctness” wrt. to category label, start and end position (yield) of all constituents (subtrees)
  - **Labelled precision:** percentage of correct subtrees in the parser output
  - **Labelled recall:** percentage of correct subtrees in the gold standard (test corpus)

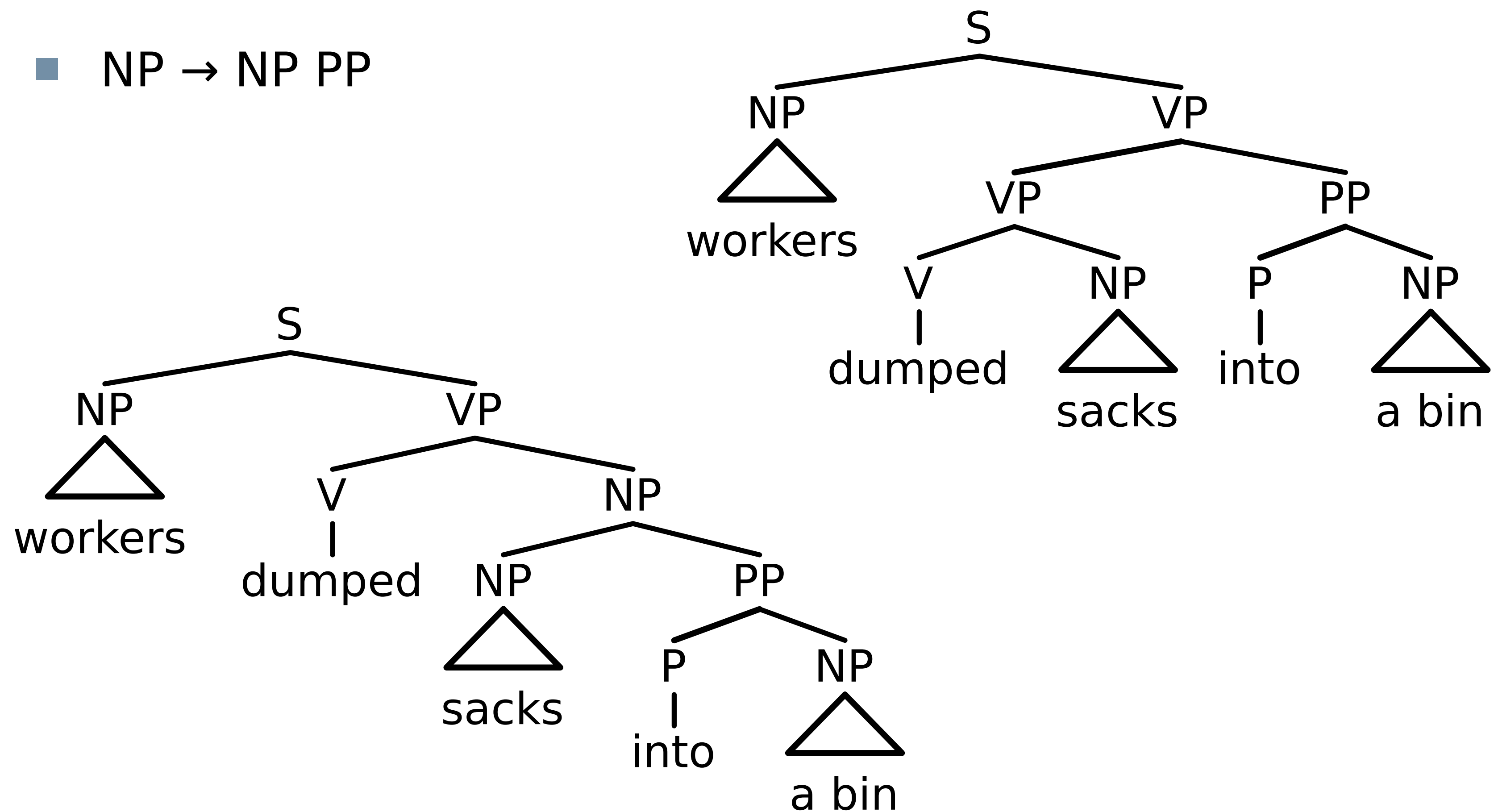
# Problems

- The probability of a (sub) tree is indipendant of
  - the context in which the tree occurs
  - the node(s) that dominates the tree
- **Problems:** we *want* to capture …
  - Lexical dependencies
  - Structural dependencies



# Lexical Dependencies

- The two trees differ only in one rule:
  - VP → VP PP
  - NP → NP PP

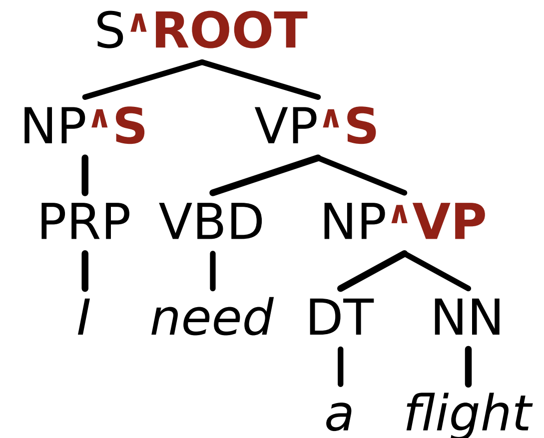


# Structural dependencies

- **Structural independencies:**
  - The (probability of an) application of a rule is independent of all other rules in the derivation tree
  - NP → Pronoun vs. NP → Det Noun
    same probabilities for all occurrences of NP
- **But …** (Francis &al, 1999)
  - Subject-NP: 91% pronouns, 9% non-pronouns
  - Object-NP: 34% pronouns, 66% non-pronouns
  - (Switchboard corpus, spoken language)
- ⇒ Parent annotation

# Structural dependencies

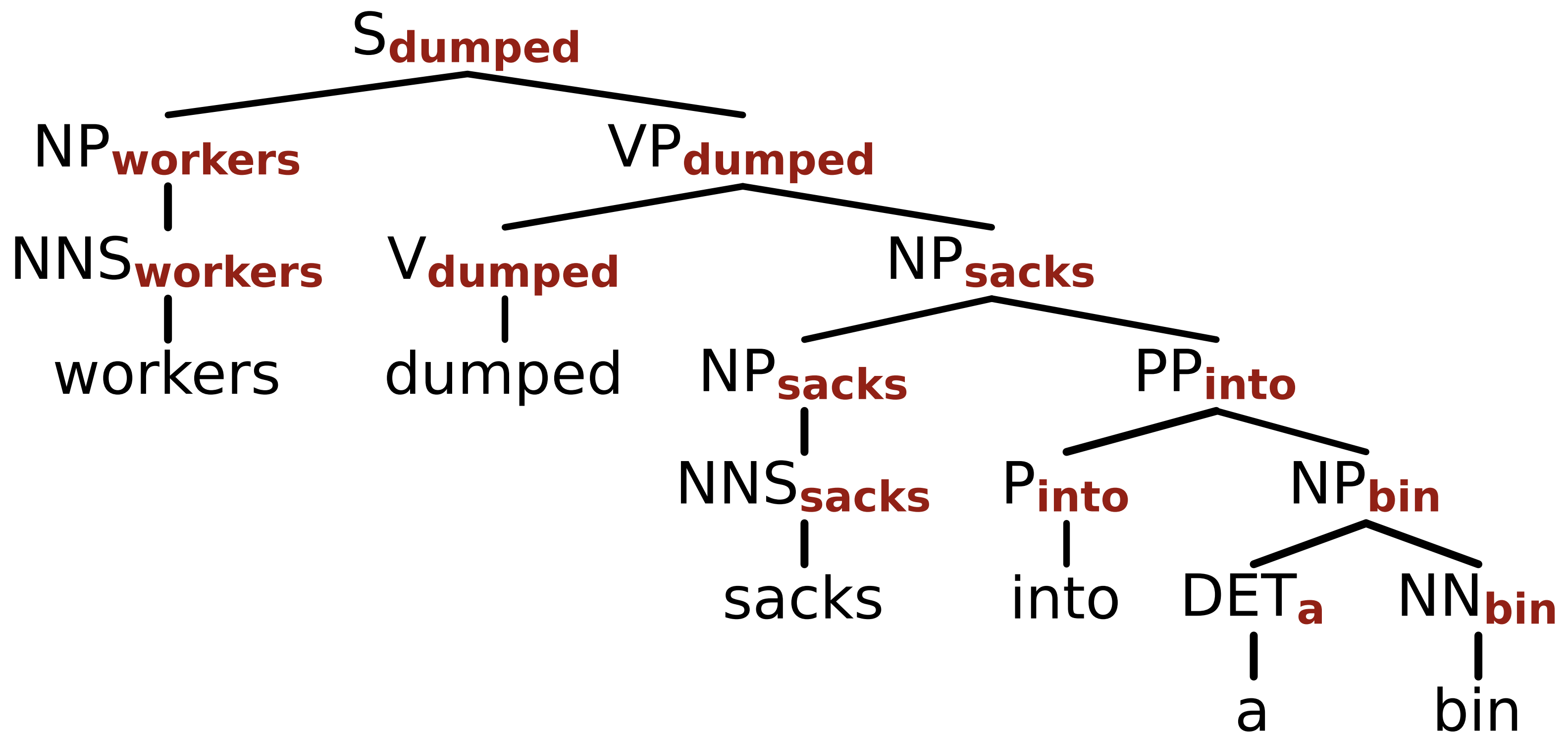
- **Parent Annotation:** nodes are annotated with the label of their parent nodes
- Similar effect compared to conditional probabilities
  - P(NP^S → PRP)
  - P(NP → PRP | S)




- Compare:
  - P(NP-SBJ → PRP) – no correspondence to conditional probabilities

# Lexical dependencies

- **Lexicalized parsing:** annotate nodes with their lexical heads



# Computational Linguistics

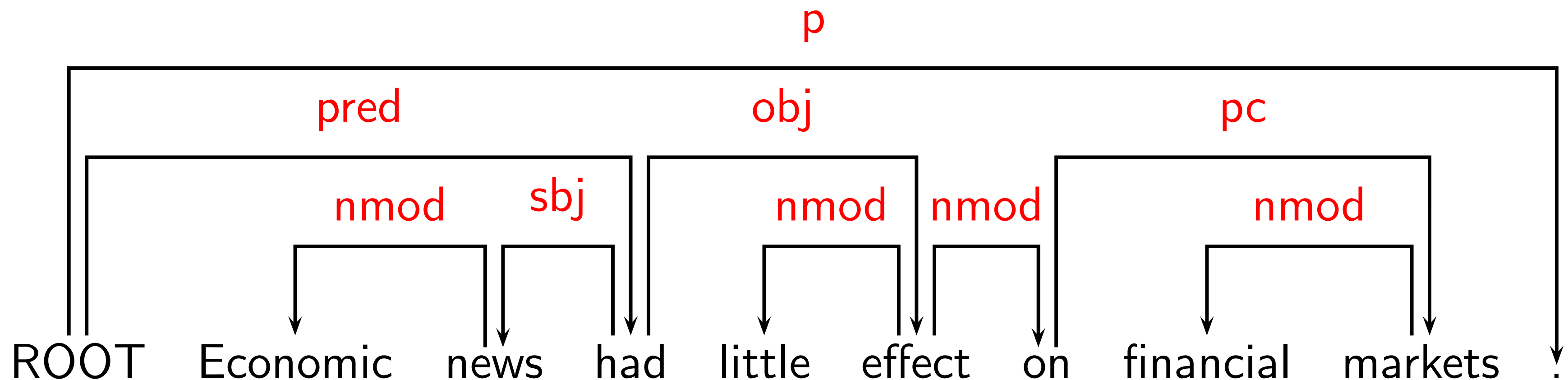
## Dependency-based Parsing

Dietrich Klakow & Stefan Thater
FR 4.7 Allgemeine Linguistik (Computerlinguistik)
Universität des Saarlandes

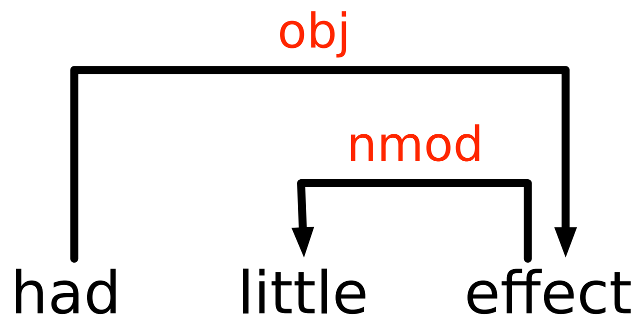
Summer 2014

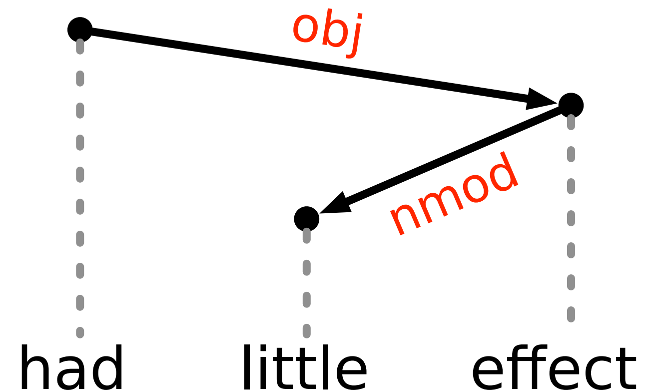
# Dependency Trees

- **Basic idea:**
  - Syntactic structure = lexical items linked by relations
  - Syntactic structures are usually trees (… but not always)
- Relations H → D
  - H is the head (or governor)
  - D is the dependent



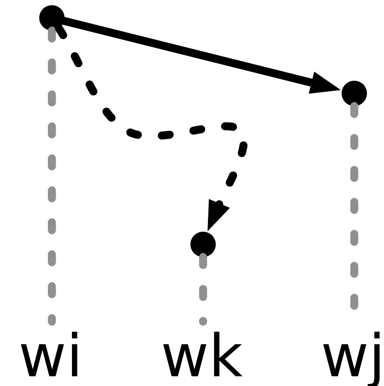
# Dependency Trees – Notation



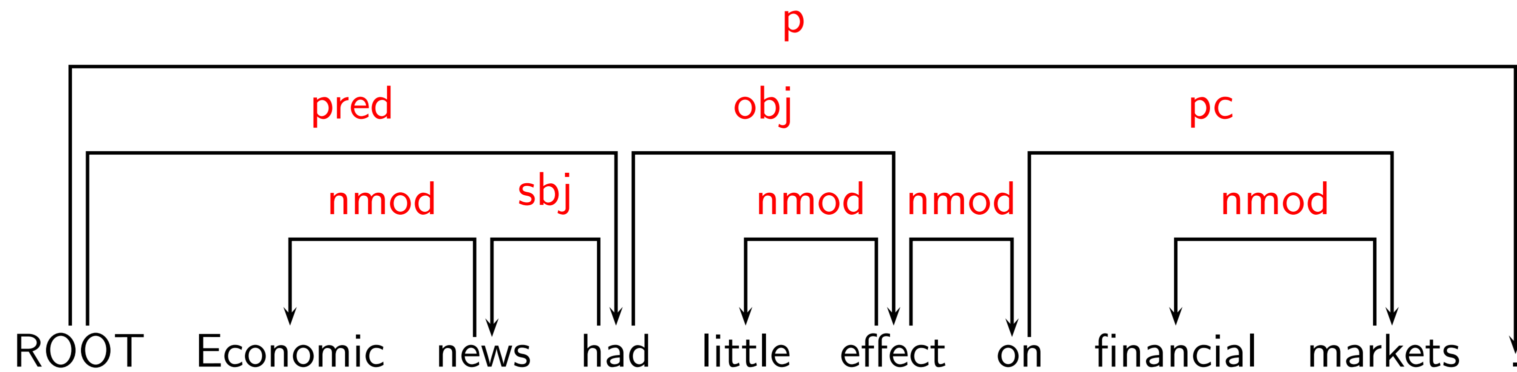



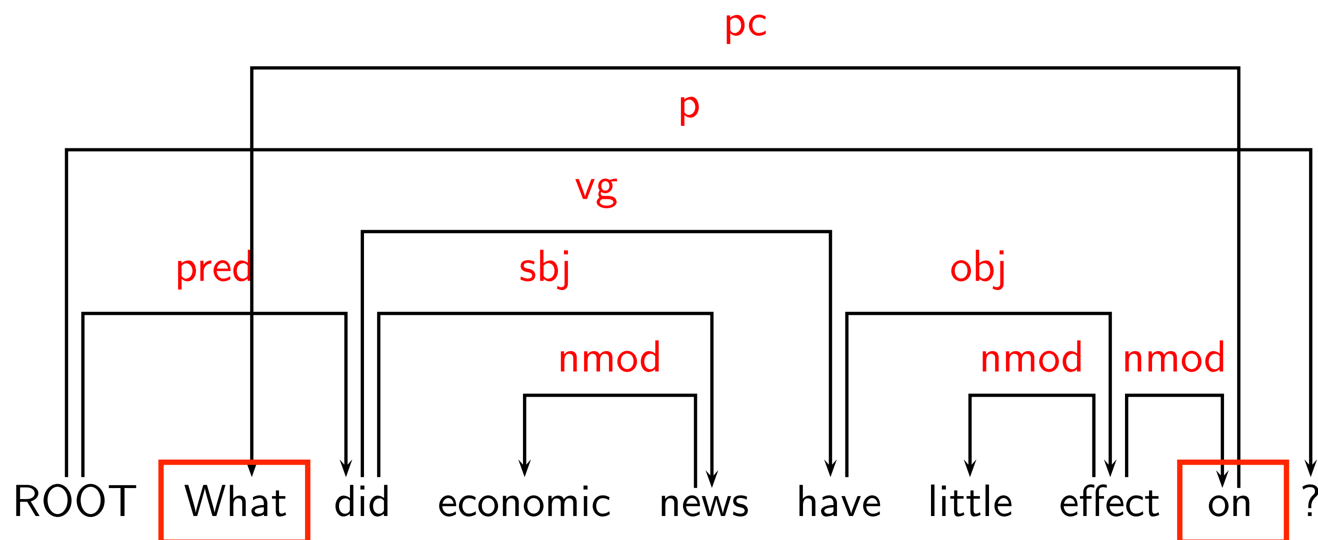
# Projectivity

- A dependency graph G is projective iff
  - if $w_i \rightarrow w_j$, then $w_i \rightarrow^* w_k$ for all $w_i < w_k < w_j$ or $w_j < w_k < w_i$
  - if $w_i$ is the head of $w_j$, then there must be a directed path from $w_i$ to $w_k$, for all $w_k$ between $w_i$ and $w_j$.
- We need non-projectivity for
  - long distance dependencies
  - free word order



# Projectivity



# Transitions („Arc-Standard“)

- Left-Arc(r)

$$\frac{\langle [\ldots, w_i], [w_j, \ldots], A \rangle}{\langle [\ldots], [w_j, \ldots], A \cup \{(w_j, r, w_i)\} \rangle} \quad i \neq 0, \neg \exists k \exists l' \, (w_k, l', w_i) \in A$$

- Right-Arc(r)

$$\frac{\langle [\ldots, w_i], [w_j, \ldots], A \rangle}{\langle [\ldots], [w_i, \ldots], A \cup \{(w_i, r, w_j)\} \rangle} \quad \neg \exists k \exists l' \, (w_k, l', w_j) \in A$$

- Shift

$$\frac{\langle [\ldots], [w_i, \ldots], A \rangle}{\langle [\ldots, w_i], [\ldots], A \rangle}$$

# Non-projective Parsing

- Configurations $\langle L_1, L_2, Q, A \rangle$
  - $L_1$, $L_2$ are stacks of partially processed nodes
  - $Q$ = a queue of unprocessed input tokens
  - $A$ = a set of dependency arcs
- Initial configuration for input $w_1 \ldots w_n$
  - $\langle [w_0], [], [w_1, \ldots, w_n], \{\} \rangle$, $w_0$ = ROOT
- Terminal configuration:
  - $\langle [w_0, w_1, \ldots, w_n], [], [], A \rangle$

# Transitions

- Left-Arc(l)

$$\frac{\langle [\ldots, w_i], [\ldots], [w_j, \ldots], A \rangle}{\langle [\ldots], [w_i, \ldots], [w_j, \ldots], A \cup \{(w_j, l, w_i)\} \rangle}$$

$i \neq 0$

$\neg \exists k \exists l' \, (w_k, l', w_i) \in A$

$\neg \, w_i \rightarrow^* w_j$

- Right-Arc(l)

$$\frac{\langle [\ldots, w_i], [\ldots], [w_j, \ldots], A \rangle}{\langle [\ldots], [w_i, \ldots], [w_j, \ldots], A \cup \{(w_i, l, w_j)\} \rangle}$$

$\neg \exists k \exists l' \, (w_k, l', w_j) \in A$

$\neg \, w_i \rightarrow^* w_j$

# Transitions

- No-Arc

$$\frac{\langle [\ldots, w_i], [\ldots], [\ldots], A \rangle}{\langle [\ldots], [w_i, \ldots], [\ldots], A \rangle}$$

- Shift

$$\frac{\langle [\ldots]_{L1}, [\ldots]_{L2}, [w_i, \ldots], A \rangle}{\langle [\ldots]_{L1} \bullet [\ldots, w_i]_{L2}, [], [\ldots], A \rangle}$$

- $L_1 \bullet L_2$ = the concatenation of $L_1$ and $L_2$