Computational Linguistics
Lecture 3 – Parsing

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

Summer 2013

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
[…]

Context-free Grammars

- Context-free grammar G = ⟨N, T, R, S⟩
  - Nonterminal symbols N
  - Terminal symbols T
  - Start symbol S ∈ N
  - Finite set of production rules: R ⊆ N x (N ∪ T)*

Derivations

- Let x, y, u, v, w, z ∈ (N ∪ T)*
- We write $x \Rightarrow_G y$ iff
  - x = uvw
  - y = uzw
  - $v \rightarrow z \in R$
- **Derivation of w_n from w_0:**
  - $w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$
- **Language generated by G = ⟨N, T, R, S⟩**
  - $L(G) = \{ w \mid S \Rightarrow_G^* w \}$
  - $\Rightarrow_G^*$ is the reflexive, transitive closure of $\Rightarrow_G$

An Example

S $\Rightarrow_G$ NP VP
$\Rightarrow_G$ DET N VP
$\Rightarrow_G$ the N VP
$\Rightarrow_G$ the student VP
$\Rightarrow_G$ the student V
$\Rightarrow_G$ the student works

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" ∈ L(G)

Another Example

S $\Rightarrow_G$ NP VP
$\Rightarrow_G$ NP V
$\Rightarrow_G$ NP works
$\Rightarrow_G$ DET N works
$\Rightarrow_G$ the N works
$\Rightarrow_G$ the student works

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" ∈ L(G)

Parse trees

- Context-free grammar $G = \langle N, T, R, S \rangle$
- **Parse trees** are trees where
  - inner nodes are labeled with symbols $\in N$
  - leaf nodes are labeled with symbols $\in T \cup \{\varepsilon\}$
  - if v is a node with label A and its child nodes $v_1, \ldots, v_n$ are labeled with $A_1, \ldots, A_n$, then $A \to A_1 \ldots A_1$ is a rule of G
  - if v is a leaf node with label ε, then v is the only child of its parent node

Leftmost derivation

- **Leftmost derivation:** replace the leftmost nonterminal symbol in each step of the derivation
- $x \Rightarrow_L y$ iff there are $A \in N$, $a, b \in (N \cup T)^*$, $w \in T^*$ such that
  - $x = wAb$
  - $y = wab$
  - $A \to a \in R$
- **Rightmost derivation:** analogously

Theorem (Lewis & Papadimitriou)

- Let $G = \langle N, T, R, S \rangle$ be a context-free grammar
- The following statements are equivalent
  - $A \Rightarrow^*_G w = w_1 \ldots w_n$
  - There is a parse tree with root A and yield w
  - There is a leftmost derivation $A \Rightarrow_L^* w$
  - There is a rightmost derivation $A \Rightarrow_R^* w$

Ambiguity

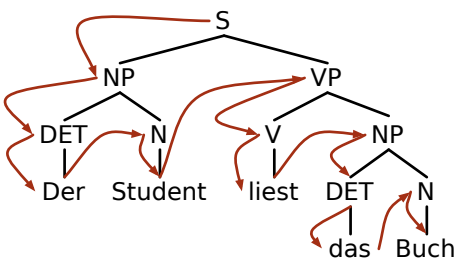
- $w = w_1 \ldots w_n$ may have two or more parse trees.
- The grammar is said to be ambiguous in this case.
- Otherwise, we say that the grammar is unambiguous.

Recognizer & Parser

- **Recognizer**
  - Is $w = w_1 \ldots w_n \in L(G)$?
- **Parser**
  - What are the parse trees of $w = w_1 \ldots w_n$?

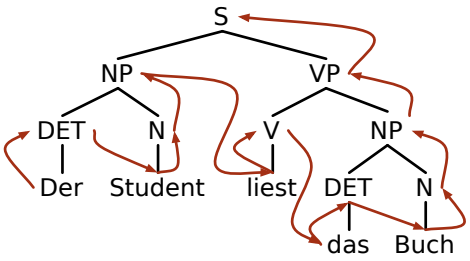
Basic Parsing Strategies

- **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 $w_1 \dots w_n$:
  - $\langle [], [w_1, \dots, w_n] \rangle$
- **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

stack

the input that still needs to be processed

Shift

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

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:**
  - $\langle [A_1, \ldots, A_{j-1}, B], [w_i, \ldots, w_n] \rangle$

An Example

$\langle [], [\textit{the student works}] \rangle$

$\Rightarrow_{\text{shift}}$ $\langle [\textit{the}], [\textit{student works}] \rangle$

$\Rightarrow_{\text{red}}$ $\langle [\text{DET}], [\textit{student works}] \rangle$

$\Rightarrow_{\text{shift}}$ $\langle [\text{DET } \textit{student}], [\textit{works}] \rangle$

$\Rightarrow_{\text{red}}$ $\langle [\text{DET N}], [\textit{works}] \rangle$

$\Rightarrow_{\text{red}}$ $\langle [\text{NP}], [\textit{works}] \rangle$

$\Rightarrow_{\text{shift}}$ $\langle [\text{NP } \textit{works}], [] \rangle$

$\Rightarrow_{\text{red}}$ $\langle [\text{NP V}], [] \rangle$

$\Rightarrow_{\text{red}}$ $\langle [\text{NP VP}], [] \rangle$

$\Rightarrow_{\text{red}}$ $\langle [\text{S}], [] \rangle$

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*

Shift or Reduce?

- How can we decide whether we should perform a shift or a reduce operation?
  - For certain (unamabiguous) grammars, it is possible to decide this automatically
  - In general ⇒ Search

Python

```
rules = [( 'S', ['NP', 'VP']), ('NP', ['DET', 'N']), ...]

def shift(stack, sent):
   return (stack + [sent[0]], sent[1:])

def reduce(stack, sent, lhs, rhs):
   return (stack[:-len(rhs)] + [lhs], sent)

def matches(stack, rhs):
    for (s, r) in zip(stack[-len(rhs):], rhs):
        if s != r:
            return False
    return True
```


Python

```
def recognize(sent):
    agenda = [([], sent)]
    while agenda:
        (stack, sent) = agenda.pop()
        if sent == [] and stack == ['S']:
            return True
        if sntnc != []:
            agenda.append(shift(stack, sent))
        for (lhs, rhs) in rules:
            if len(stack) >= len(rhs):
                if matches(stack, rhs):
                    agenda.append(reduce(stack, sent, lhs, rhs))
    return False
```


Example – *The student works*

⟨stack, sent⟩	agenda
–	⟨[] [the student works]⟩
⟨[] [the student works]⟩	⟨[the] [student works]⟩
⟨[the] [student works]⟩	⟨[DET] [student works]⟩ ⟨[the student] [works]⟩
⟨[DET] [student works]⟩	⟨[DET student] [works]⟩ ⟨[the student] [works]⟩
⟨[DET student] [works]⟩	⟨[DET N] [works]⟩ ⟨[DET student works] []⟩ ⟨[the student] [works]⟩
⟨[NP] [works]⟩	⟨[NP] [works]⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ …
⟨[NP works] []⟩	⟨[NP works] []⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ …
⟨[NP V] []⟩	⟨[NP V] []⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ …
⟨[NP VP] []⟩	⟨[NP VP] []⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ …
⟨[S] []⟩	⟨[S] []⟩ ⟨[DET N works] []⟩ ⟨[DET student works] []⟩ …

Example – *The student reads …*

- [⇒ Handout]

Problematic Rules

- Bottom-up parsers cannot deal with certain types of grammars (the parser may not terminate)
- Rules of the form $A \rightarrow \varepsilon$
  - $\langle [A_1, \ldots, A_k], [w_i, \ldots, w_n] \rangle$
  - $\langle [A_1, \ldots, A_k, A], [w_i, \ldots, w_n] \rangle$ (reduce)
  - $\langle [A_1, \ldots, A_k, A, A], [w_i, \ldots, w_n] \rangle$ (reduce)
  - $\langle [A_1, \ldots, A_k, A, A, A], [w_i, \ldots, w_n] \rangle$ (reduce)
  - […]

Problematic Rules

- Bottom-up parsers cannot deal with certain types of grammars (the parser may not terminate)
- Cyclic rules: $A \rightarrow B$, $B \rightarrow A$
  - $\langle [A_1, \ldots, A_k, A], [w_i, \ldots, w_n] \rangle$
  - $\langle [A_1, \ldots, A_k, B], [w_i, \ldots, w_n] \rangle$ (reduce)
  - $\langle [A_1, \ldots, A_k, A], [w_i, \ldots, w_n] \rangle$ (reduce)
  - $\langle [A_1, \ldots, A_k, B], [w_i, \ldots, w_n] \rangle$ (reduce)
  - […]

Another Problem …

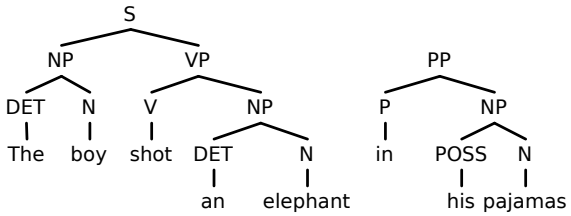
The boy shot an elephant in …

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

⇒* ⟨[NP VP], [*in his pajamas*]⟩

⇒ ⟨[S], [*in his pajamas*]⟩

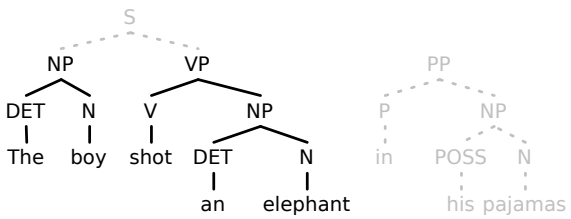
⇒* ⟨[S PP], []⟩ ⇒ Failure, Backtracking



The boy shot an elephant in …

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

⇒* ⟨[NP VP], [*in his pajamas*]⟩

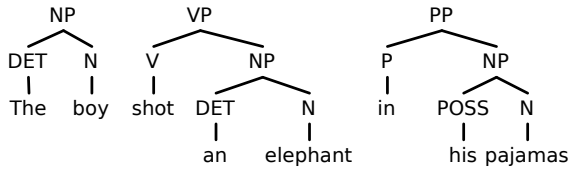


The boy shot an elephant in ...

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

⇒* ⟨[NP VP], [*in his pajamas*]⟩

⇒* ⟨[NP VP PP], []⟩



The boy shot an elephant in ...

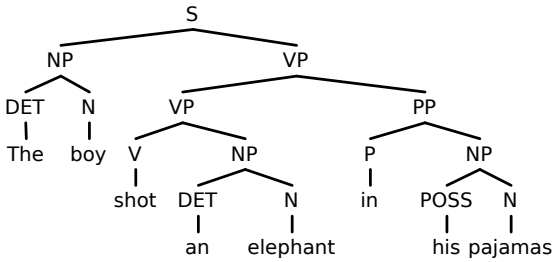
⟨[], [*the boy shot an elephant in his pajamas*]⟩

⇒* ⟨[NP VP], [*in his pajamas*]⟩

⇒* ⟨[NP VP PP], []⟩

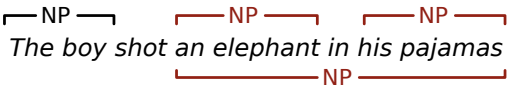
⇒* ⟨[NP VP], []⟩

⇒* ⟨[S], []⟩



Dynamic Programming

- **Context-free grammar:** whether or not a rule can be applied does not depend on the context.



- **Chart-Parsing:** store intermediate results for already analysed constituents in a “chart”
- **Charts** are compact representations of all possible analyses (“parse forest”)

Chart-Parsing

- **Chart-Parsing:** store intermediate results for already analysed constituents in a “chart”
- **Charts** are compact representations of all possible analyses (“parse forest”)
- Charts can contain
  - complete constituents
  - hypotheses for possible constituents
- Many different chart-parsers:
  - Cocke-Younger-Kasami, Earley, …

Charts as Matrices

- **A ∈ T[i, j]** iff A ⇒* w_{i+1} … 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*

Cocke-Younger-Kasami

- The algorithm by Cocke, Younger, Kasami (CYK) is a simple chart-based bottom-up parser
- **Restriction:** the algorithm can be applied to grammars in Chomsky normal form only:
  - A → w (w terminal symbol)
  - A → B C (B and C nonterminal symbols)
  - S → ε (S start symbol, only if ε ∈ L)
- **Note:** we will assume here that ε ∉ L, thus the grammar will not contain rules S → ε

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
```


An Example

- [⇒ blackboard]

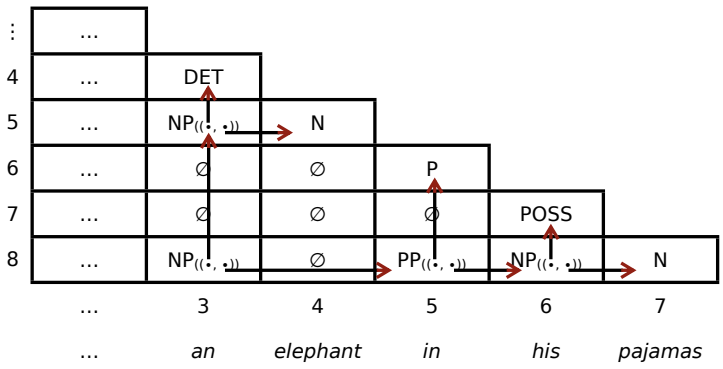
Properties

- **Correct:**
  If $S \in T[0, n]$, then $S \Rightarrow^* w_1 \ldots w_n$
- **Complete:**
  If $S \Rightarrow^* w_1 \ldots w_n$, then $S \in T[0, n]$
- **Runtime:**
  Polynomial in the input length: $O(n^3)$

Recognizer → Parser

- The recognizer can be extended to a parser if we store, for each category A, a list of pointers to other entries in the chart that have been used to derive A

CYK (Parser)



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] ∪ { A | A → B C, B ∈ T[i,k], C ∈ T[k,j] }
  - ⇒ can be implemented in different ways
- **Method 1**
  - Iterate over all rules A → B C
  - Check if B ∈ T[i,k] and C ∈ T[k,j]
- **Method 2**
  - Iterate over all B ∈ T[i,k]
  - Iterate over all rules A → B C
  - Check if C ∈ T[k, j]

Implementierungsvarianten

- T[i,j] = T[i,j] ∪ { A | A → B C, B ∈ T[i,k], C ∈ T[k,j] }
  - ⇒ can be implemented in different ways
- **Method 3**
  - Iterate over all C ∈ T[k,j]
  - Iterate over all rules A → B C
  - Check if A ∈ T[i,k]
- **Method 4**
  - Iterate over all B ∈ T[i,k] and C ∈ T[k,j]
  - Check if a rule A → B C exists

Song &al. (EMNLP 2008)

- Experiments mit CYK & Wall Street Journal
- Runtime depends on ...
  - right binarization ⇒ method 3 is most efficient
  - left binarization ⇒ method 2 is most efficient