עיבוד שפות טבעיות

שולי וינטנר

Parsing

Recognition: Given a (context-free) grammar G and a string of words w, determine whether $w \in L(G)$.

Parsing: If $w \in L(G)$, produce the (tree) structure that is assigned by G to w.

Parsing

General requirements for a parsing algorithm:

- Generality: the algorithm must be applicable to *any* grammar
- Completeness: the algorithm must produce *all* the results in case of ambiguity
- Efficiency
- Flexibility: a good algorithm can be easily modified

Parsing

Parameters that define different parsing algorithm:

Orientation: Top-down vs. bottom-up vs. mixed

Direction: Left-to-right vs. right-to-left vs. mixed (e.g., island-driven)

Handling multiple choice: Dynamic programming vs. parallel processing vs. backtracking

Search: Breadth-first or Depth-first

A bottom-up recognition algorithm

Assumptions:

- The grammar is given in Chomsky Normal Form: each rule is either of the form $A \rightarrow B\ C$ (where A, B, C are non-terminals) or of the form $A \rightarrow a$ (where a is a terminal).
- The string to recognize is $w = w_1 \cdots w_n$.
- A set of indices $\{0, 1, \ldots, n\}$ is defined to point to positions between the input string's words:

  0 the 1 cat 2 in 3 the 4 hat 5

An example grammar

$D \rightarrow$ *the*	$NP \rightarrow D\ N$
$N \rightarrow$ *cat*	$PP \rightarrow P\ NP$
$N \rightarrow$ *hat*	$NP \rightarrow NP\ PP$
$P \rightarrow$ *in*	

Example sentences:

the cat in the hat

the cat in the hat in the hat

The CYK algorithm

Bottom-up, chart-based recognition algorithm for grammars in CNF

To recognize a string of length n, uses a *chart*: a bi-dimensional matrix of size $n \times (n+1)$

Invariant: a non-terminal A is stored in the $[i, j]$ entry of the chart iff $A \Rightarrow w_{i+1} \cdots w_j$

Consequently, the chart is triangular. A word w is recognized iff the start symbol S is in the $[0, n]$ entry of the chart

The idea: build all constituents up to the i-th position before constructing the $i+1$ position; build smaller constituents before constructing larger ones

The CYK algorithm

```
for j := 1 to n do
   for all rules A → w_j do
     chart[j-1,j] := chart[j-1,j] ∪ {A}
   for i := j-2 downto 0 do
     for k := i+1 to j-1 do
       for all B ∈ chart[i,k] do
         for all C ∈ chart[k,j] do
           for all rules A → B C do
             chart[i,j] := chart[i,j] ∪ {A}
if S ∈ chart[0,n] then accept else reject
```

The CYK algorithm

Extensions:

- Parsing in addition to recognition
- Support for ϵ-rules
- General context-free grammars (not just CYK)

Parsing schemata

To provide a unified framework for discussing various parsing algorithms we use *parsing schemata*, which are generalized schemes for describing the principles behind specific parsing algorithms. This is a generalization of the *parsing as deduction* paradigm.

A parsing schema consists of four components:

- a set of items
- a set of axioms
- a set of deduction rules
- a set of goal items

Parsing schema: CYK

Given a grammar $G = \langle \Sigma, V, S, P \rangle$ and a string $w = w_1 \cdots w_n$:

Items: $[i, A, j]$ for $A \in V$ and $0 \leq i, j \leq n$
(state that $A \stackrel{*}{\Rightarrow} w_{i+1} \cdots w_j$)

Axioms: $[i, A, i+1]$ when $A \rightarrow w_{i+1} \in P$

Goals: $[0, S, n]$

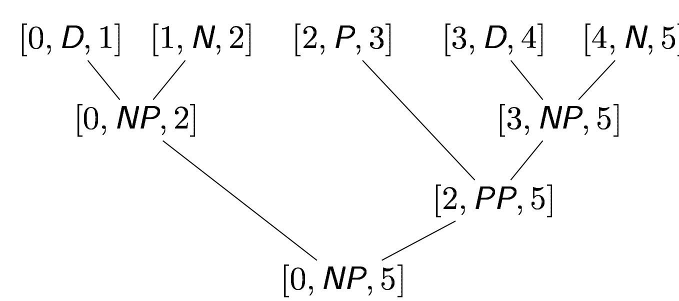
Inference rules:

$$\frac{[i, B, j] \quad [j, C, k]}{[i, A, k]} \quad A \rightarrow B\ C$$

CYK parsing schema: deduction example

D → *the*	*NP* → *D N*
N → *cat*	*PP* → *P NP*
N → *hat*	*NP* → *NP PP*
P → *in*	

0 the 1 cat 2 in 3 the 4 hat 5



Parsing: bottom-up schema (Shift–Reduce)

Items: $[\alpha\bullet, j]$ (state that $\alpha w_{j+1} \cdots w_n \stackrel{*}{\Rightarrow} w_1 \cdots w_n$)

Axioms: $[\bullet, 0]$

Goals: $[S\bullet, n]$

Inference rules:

Shift $$\frac{[\alpha\bullet, j]}{[\alpha w_{j+1}\bullet, j+1]}$$

Reduce $$\frac{[\alpha\gamma\bullet, j]}{[\alpha B\bullet, j]} \quad B \rightarrow \gamma$$

Bottom-up deduction: example

Parsing: top-down schema

Item form: $[\bullet\beta, j]$ (state that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_j\beta$)

Axioms: $[\bullet S, 0]$

Goals: $[\bullet, n]$

Inference rules:

Scan $$\frac{[\bullet w_{j+1}\beta, j]}{[\bullet\beta, j+1]}$$

Predict $$\frac{[\bullet B\beta, j]}{[\bullet\gamma\beta, j]} \quad B \rightarrow \gamma$$

Top-down deduction: example

Input: 0 the 1 cat 2 in 3 the 4 hat 5

$[\bullet NP, 0]$	*axiom*
$[\bullet NP\ PP, 0]$	*predict* $NP \rightarrow NP\ PP$
$[\bullet D\ N\ PP, 0]$	*predict* $NP \rightarrow D\ N$
$[\bullet the\ N\ PP, 0]$	*predict* $D \rightarrow the$
$[\bullet N\ PP, 1]$	*scan*
$[\bullet cat\ PP, 1]$	*predict* $N \rightarrow cat$
$[\bullet PP, 2]$	*scan*
$[\bullet P\ NP, 2]$	*predict* $PP \rightarrow P\ NP$
$[\bullet in\ NP, 2]$	*predict* $P \rightarrow in$
$[\bullet NP, 3]$	*scan*
$[\bullet D\ N, 3]$	*predict* $NP \rightarrow D\ N$
$[\bullet the\ N, 3]$	*predict* $D \rightarrow the$
$[\bullet N, 4]$	*scan*
$[\bullet hat, 4]$	*predict* $N \rightarrow hat$
$[\bullet, 5]$	*scan*

Top-down parsing: algorithm

```
Parse(β,j) ::
  if β = w_{j+1} · β' then return parse(β', j+1)
  else if β = B · β' then
    for every rule B → γ ∈ P
      if Parse(γ · β', j) then return true
  return false

if Parse(S,0) then accept else reject
```


Top-down vs. Bottom-up parsing

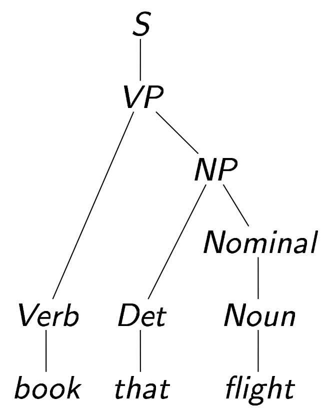
Two inherent constraints:

1. The root of the tree is *S*
2. The yield of the tree is the input word

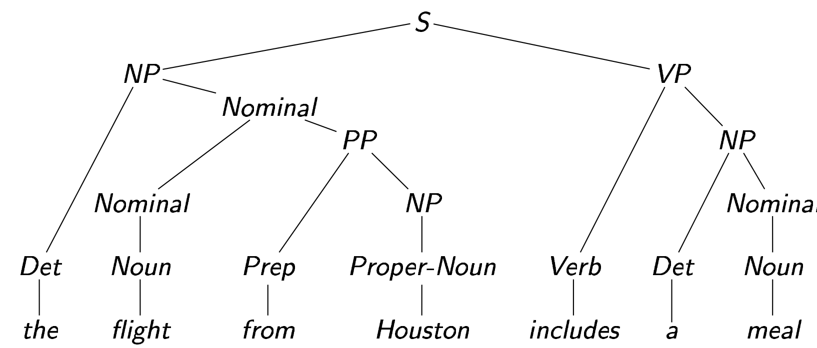
An example grammar

$S \rightarrow NP\ VP$	*Det* → *that* | *this* | *a*
$S \rightarrow Aux\ NP\ VP$	*Noun* → *book* | *flight* | *meal*
$S \rightarrow VP$	*Verb* → *book* | *include* | *includes*
$VP \rightarrow Verb$	*Prep* → *from* | *to* | *on*
$VP \rightarrow Verb\ NP$	*Proper-Noun* → *Houston* | *TWA*
NP → *Det Nominal*	*Aux* → *does*
NP → *Proper-Noun*	
Nominal → *Noun*	
Nominal → *Noun Nominal*	
Nominal → *Nominal PP*	
PP → *Prep NP*	

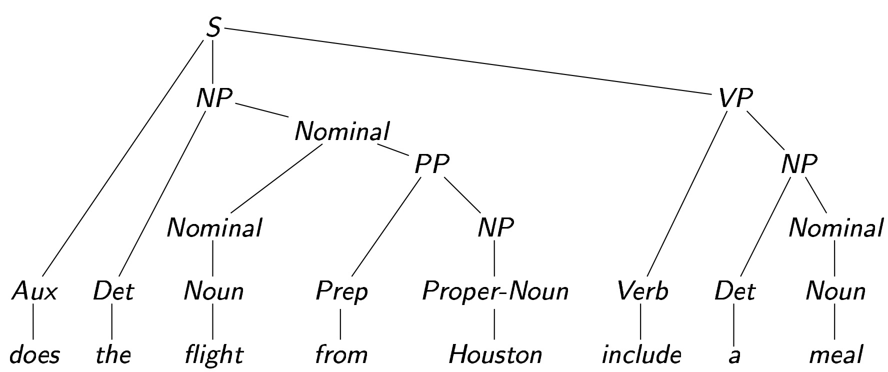
An example derivation tree



An example derivation tree



An example derivation tree



Top-down vs. Bottom-up parsing

When expanding the top-down search space, which local trees are created?

Top-down vs. Bottom-up parsing

To reduce "blind" search, add bottom-up filtering.

Observation: when trying to Parse(β,j), where $\beta = B\gamma$, the parser succeeds only if $B \overset{*}{\Rightarrow} w_{j+1}\beta$.

Definition: A word w is a **left-corner** of a non-terminal B iff $B \overset{*}{\Rightarrow} w\beta$ for some β.

Top-down parsing with bottom-up filtering

Parse(β,j) ::
if $\beta = w_{j+1} \cdot \beta'$ then return parse($\beta', j+1$)
else if $\beta = B \cdot \beta'$ then
if w_{j+1} is a left-corner of B then
for every rule $B \rightarrow \gamma \in P$
if Parse($\gamma \cdot \beta', j$) then return true
return false

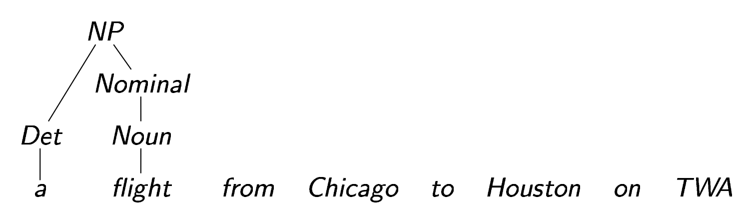
if Parse(S,0) then accept else reject

Top-down vs. Bottom-up parsing

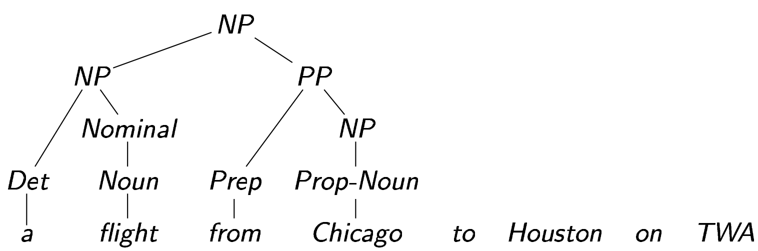
Even with bottom-up filtering, top-down parsing suffers from the following problems:

- Left recursive rules can cause non-termination: $NP \rightarrow NP\ PP$.
- Even when parsing terminates, it might take exponentially many steps.
- Constituents are computed over and over again

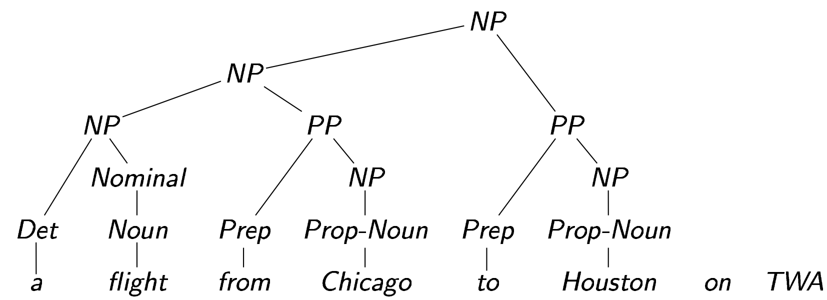
Top-down parsing: repeated generation of sub-trees



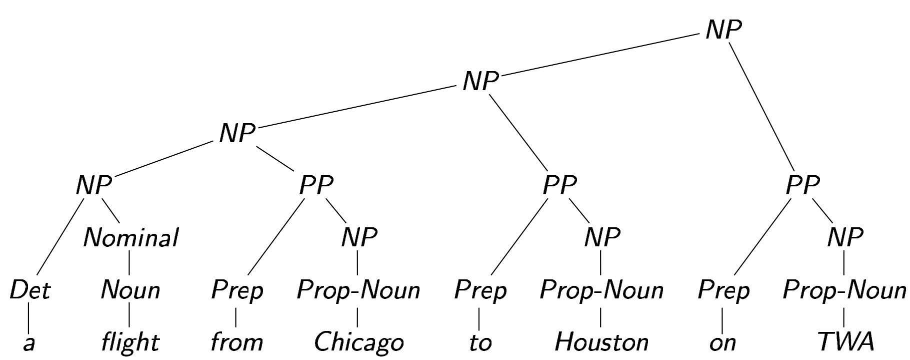
Top-down parsing: repeated generation of sub-trees



Top-down parsing: repeated generation of sub-trees



Top-down parsing: repeated generation of sub-trees



Top-down parsing: repeated generation of sub-trees

Reduplication:

Constituent	#
a flight	4
from Chicago	3
to Houston	2
on TWA	1
a flight from Chicago	3
a flight from Chicago to Houston	2
a flight from Chicago to Houston on TWA	1

Top-down vs. Bottom-up parsing

When expanding the bottom-up search space, which local trees are created?

Top-down vs. Bottom-up parsing

Bottom-up parsing suffers from the following problems:

- All possible analyses of every substring are generated, even when they can never lead to an *S*, or can never combine with their neighbors
- ϵ-rules can cause performance degradation
- Reduplication of effort

Earley's parsing algorithm

- Dynamic programming: partial results are stored in a chart
- Combines top-down predictions with bottom-up scanning
- No reduplication of computation
- Left-recursion is correctly handled
- ϵ-rules are handled correctly
- Worst-case complexity: $O(n^3)$

Earley's parsing algorithm

Basic concepts:

Dotted rules: if $A \rightarrow \alpha\beta$ is a grammar rule then $A \rightarrow \alpha \bullet \beta$ is a dotted rule.

Edges: if $A \rightarrow \alpha \bullet \beta$ is a dotted rule and i, j are indices into the input string then $[i, A \rightarrow \alpha \bullet \beta, j]$ is an edge. An edge is **passive** (or **complete**) if $\beta = \epsilon$, **active** otherwise.

Actions: The algorithm performs three operations: *scan*, *predict* and *complete*.

Earley's parsing algorithm

scan: read an input word and add a corresponding complete edge to the chart.

predict: when an active edge is added to the chart, predict all possible edges that can follow it

complete: when a complete edge is added to the chart, combine it with appropriate active edges

Earley's parsing algorithm

rightsisters: given an active edge $A \to \alpha \bullet B\beta$, return all dotted rules $B \to \bullet\gamma$

leftsisters: given a complete edge $B \to \gamma\bullet$, return all dotted edges $A \to \alpha \bullet B\beta$

combination:

$$[i, A \to \alpha \bullet B\beta, k] * [k, B \to \gamma\bullet, j] = [i, A \to \alpha B \bullet \beta, j]$$

Parsing: Earley deduction

Item form: $[i, A \to \alpha \bullet \beta, j]$ (state that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_i A\gamma$, and also that $\alpha \stackrel{*}{\Rightarrow} w_{i+1} \cdots w_j$)

Axioms: $[0, S' \to \bullet\ S, 0]$

Goals: $[0, S' \to S\ \bullet, n]$

Parsing: Earley deduction

Inference rules:

Scan
$$\frac{[i, A \to \alpha \bullet w_{j+1}\beta, j]}{[i, A \to \alpha w_{j+1} \bullet \beta, j+1]}$$

Predict
$$\frac{[i, A \to \alpha \bullet B\beta, j]}{[j, B \to \bullet\gamma, j]} \quad B \to \gamma$$

Complete
$$\frac{[i, A \to \alpha \bullet B\beta, k] \quad [k, B \to \gamma\bullet, j]}{[i, A \to \alpha B \bullet \beta, j]}$$

Earley's parsing algorithm

```
Parse ::
   enteredge([0,S' → • S,0])
   for j := 1 to n do
     for every rule A → w_j do
       enteredge([j-1,A → w_j•,j])

if S' → S • ∈ C[0,n] then accept else reject
```


Earley's parsing algorithm

```
enteredge(i,edge,j) ::
   if edge ∉ C[i,j] then /* occurs check */
     C[i,j] := C[i,j] ∪ {edge}
     if edge is active then /* predict */
       for edge' ∈ rightsisters(edge) do
         enteredge([j,edge',j])
     if edge is passive then /* complete */
       for edge' ∈ leftsisters(edge) do
         for k such that edge' ∈ C[k,i] do
           enteredge([k,edge'*edge,j])
```