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 algorithms:

**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
An example grammar

\[
\begin{align*}
D & \rightarrow \text{the} & NP & \rightarrow D \ N \\
N & \rightarrow \text{cat} & PP & \rightarrow P \ NP \\
N & \rightarrow \text{hat} & NP & \rightarrow NP \ PP \\
P & \rightarrow \text{in} & \end{align*}
\]

Example sentences:

the cat in the hat
the cat in the hat in the hat
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
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 $A \rightarrow w_j$ do
        chart[j-1,j] := chart[j-1,j] $\cup \{A\}$
    for i := j-2 downto 0 do
        for k := i+1 to j-1 do
            for all $B \in$ chart[i,k] do
                for all $C \in$ chart[k,j] do
                    for all rules $A \rightarrow BC$ do
                        chart[i,j] := chart[i,j] $\cup \{A\}$
        if $S \in$ chart[0,n] then accept else reject
The CYK algorithm

Extensions:

- Parsing in addition to recognition
- Support for $\epsilon$-rules
- General context-free grammars (not just CNF)
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 \( A \xrightarrow{*} w_{i+1} \cdots w_j \))

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

**Goals:** \([0, S, n]\)

**Inference rules:**

\[
\begin{array}{c}
[i, B, j] \quad [j, C, k] \\
\hline
[i, A, k] \\
A \rightarrow B \ C
\end{array}
\]
CYK parsing schema: deduction example

\[ D \rightarrow \text{the} \quad NP \rightarrow D \ N \]
\[ N \rightarrow \text{cat} \quad PP \rightarrow P \ NP \]
\[ N \rightarrow \text{hat} \quad NP \rightarrow NP \ PP \]
\[ P \rightarrow \text{in} \]

0 the 1 cat 2 in 3 the 4 hat 5

\[
\begin{array}{cccccc}
[0, NP, 2] & & [3, NP, 5] & & \\
& [2, PP, 5] & & \\
& & [0, NP, 5] & & \\
\end{array}
\]
Parsing: bottom-up schema (Shift–Reduce)

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

Axioms: $[\bullet, 0]$

Goals: $[S \bullet, n]$

Inference rules:

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

\[
\text{Reduce} \quad \frac{[\alpha \gamma \bullet, j]}{[\alpha B \bullet, j]} \quad B \rightarrow \gamma
\]
Bottom-up deduction: example
Parsing: top-down schema

Item form: \([\bullet \beta, j]\) (state that \(S \Rightarrow^* w_1 \ldots w_j \beta\))

Axioms: \([\bullet S, 0]\)

Goals: \([\bullet, n]\)

Inference rules:

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

\[
\text{Predict} \quad \frac{[\bullet B \beta, j]}{[\bullet \gamma \beta, j]} \quad B \rightarrow \gamma
\]
Top-down deduction: example

Input: 0 the 1 cat 2 in 3 the 4 hat 5
[●NP, 0] axiom
[●NP PP, 0] predict NP → NP PP
[●D N PP, 0] predict NP → D N
[●the N PP, 0] predict D → the
[●N PP, 1] scan
[●cat PP, 1] predict N → cat
[●PP, 2] scan
[●P NP, 2] predict PP → P NP
[●in NP, 2] predict P → in
[●NP, 3] scan
[●D N, 3] predict NP → D N
[●the N, 3] predict D → the
[●N, 4] scan
[●hat, 4] predict N → hat
[●, 5] scan
Top-down parsing: algorithm

Parse(\(\beta, j\)) ::
    if \(\beta = w_{j+1} \cdot \beta'\) then return parse(\(\beta', j + 1\))
    else if \(\beta = B \cdot \beta'\) 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

Two inherent constraints:

1. The root of the tree is $S$

2. The yield of the tree is the input word
An example grammar

\[
\begin{align*}
S &\rightarrow NP \ VP \\
S &\rightarrow Aux \ NP \ VP \\
S &\rightarrow VP \\
VP &\rightarrow Verb \\
VP &\rightarrow Verb \ NP \\
NP &\rightarrow Det \ Nominal \\
NP &\rightarrow Proper-Noun \\
Nominal &\rightarrow Noun \\
Nominal &\rightarrow Noun \ Nominal \\
Nominal &\rightarrow Nominal \ PP \\
PP &\rightarrow Prep \ NP
\end{align*}
\]

\[
\begin{align*}
Det &\rightarrow that \ | \ this \ | \ a \\
Noun &\rightarrow book \ | \ flight \ | \ meal \\
Verb &\rightarrow book \ | \ include \ | \ includes \\
Prep &\rightarrow from \ | \ to \ | \ on \\
Proper-Noun &\rightarrow Houston \ | \ TWA \\
Aux &\rightarrow does
\end{align*}
\]
An example derivation tree

S
  VP
    NP
      Nominal
        Verb
        Det
        Noun
          book
          that
          flight
An example derivation tree
An example derivation tree

S
  / \    /
 NP   VP
  |    |
 Nominal PP NP Nominal
  |    |   |   |
 Aux Det Noun Prep Proper-Noun Verb Det Noun
     the flight from Houston include a meal
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($\beta$, j), where $\beta = B\gamma$, the parser succeeds only if $B \Rightarrow^* w_{j+1}\beta$.

Definition: A word $w$ is a **left-corner** of a non-terminal $B$ iff $B \Rightarrow^* w\beta$ for some $\beta$. 
**Top-down parsing with bottom-up filtering**

\[
\text{Parse}(\beta, j) :: \\
\quad \text{if } \beta = w_{j+1} \cdot \beta' \text{ then return parse}(\beta', j + 1) \\
\quad \text{else if } \beta = B \cdot \beta' \text{ then} \\
\begin{align*}
&\quad \text{if } w_{j+1} \text{ is a left-corner of } B \text{ then} \\
&\quad \quad \text{for every rule } B \rightarrow \gamma \in P \\
&\quad \quad \quad \text{if Parse}(\gamma \cdot \beta', j) \text{ then return true} \\
&\text{return false}
\end{align*}
\]

\text{if Parse}(S, 0) \text{ 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

```
NP
  Nominal
    Det Noun
    a flight from Chicago to Houston on TWA
```
Top-down parsing: repeated generation of sub-trees

```
Det  Nominal  Prep  Prop-Noun
  a    flight   from  Chicago  to  Houston  on  TWA
```
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:

<table>
<thead>
<tr>
<th>Constituent</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>a flight</td>
<td>4</td>
</tr>
<tr>
<td>from Chicago</td>
<td>3</td>
</tr>
<tr>
<td>to Houston</td>
<td>2</td>
</tr>
<tr>
<td>on TWA</td>
<td>1</td>
</tr>
<tr>
<td>a flight from Chicago</td>
<td>3</td>
</tr>
<tr>
<td>a flight from Chicago to Houston</td>
<td>2</td>
</tr>
<tr>
<td>a flight from Chicago to Houston on TWA</td>
<td>1</td>
</tr>
</tbody>
</table>
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
- $\epsilon$-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
- $\epsilon$-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 \( A \rightarrow \alpha \cdot \beta \) is a dotted rule.

**Edges:** if \( A \rightarrow \alpha \cdot \beta \) is a dotted rule and \( i, j \) are indices into the input string then \([i, A \rightarrow \alpha \cdot \beta, j]\) 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

**rightsiseters:** given an active edge $A \rightarrow \alpha \bullet B\beta$, return all dotted rules $B \rightarrow \bullet \gamma$

**leftsisisters:** given a complete edge $B \rightarrow \gamma\bullet$, return all dotted edges $A \rightarrow \alpha \bullet B\beta$

**combination:**

$$[i, A \rightarrow \alpha \bullet B\beta, k] \ast [k, B \rightarrow \gamma\bullet, j] = [i, A \rightarrow \alpha B \bullet \beta, j]$$
Parsing: Earley deduction

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

**Axioms:** \([0, S' \rightarrow \bullet S, 0]\)

**Goals:** \([0, S' \rightarrow S \bullet, n]\)
Parsing: Earley deduction

Inference rules:

Scan

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

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

Predict

\[ [i, A \to \alpha \bullet B\beta, j] \]

\[ \frac{}{[j, B \to \bullet \gamma, j]} \]

\[ B \to \gamma \]

Complete

\[ [i, A \to \alpha \bullet B\beta, k] \quad [k, B \to \gamma \bullet, j] \]

\[ \frac{}{[i, A \to \alpha B \bullet \beta, j]} \]
Earley's parsing algorithm

Parse ::

\[
\text{enteredge}(0, S' \rightarrow \bullet S, 0)
\]

for \( j := 1 \) to \( n \) do

for every rule \( A \rightarrow w_j \) do

\[
\text{enteredge}([j-1, A \rightarrow w_j \bullet, j])
\]

if \( S' \rightarrow S \bullet \in C[0,n] \) then accept else reject
Earley's parsing algorithm

enteredge(i,edge,j) ::
    if edge $\not\in$ C[i,j] then /* occurs check */
        C[i,j] := C[i,j] $\cup$ {edge}
    if edge is active then /* predict */
        for edge’ $\in$ rightsisters(edge) do
            enteredge([j,edge’,j])
    if edge is passive then /* complete */
        for edge’ $\in$ leftsisters(edge) do
            for k such that edge’ $\in$ C[k,i] do
                enteredge([k,edge’*edge,j])