**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 $w$.

---

**An example grammar**

- $D \rightarrow \text{the}$
- $NP \rightarrow D N$
- $N \rightarrow \text{cat}$
- $PP \rightarrow P NP$
- $N \rightarrow \text{hat}$
- $NP \rightarrow NP PP$
- $P \rightarrow \text{in}$

**Example sentences:**

- the cat in the hat
- the cat in the hat in the hat

---

**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

**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
A bottom-up recognition algorithm

Assumptions:

- The grammar is given in Chomsky Normal Form; each rule is either of the form $A \to BC$ (where $A$, $B$, $C$ are non-terminals) or of the form $A \to a$ (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

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

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

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

CYK parsing schema: deduction example

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

\[
\begin{align*}
&[0, D, 1] \\
&[1, N, 2] \\
&[2, P, 3] \\
&[3, D, 4] \\
&[4, N, 5] \\
&[0, NP, 2] \\
&[3, NP, 5] \\
&[2, PP, 5] \\
&[0, NP, 5]
\end{align*}
\]

Parsing schema: CYK

Given a grammar \( G = (\Sigma, V, S, P) \) 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 \( A \Rightarrow w_{i+1} \cdots w_j \))
- **Axioms**: \([i, A, i+1]\) when \( A \rightarrow w_{i+1} \in P \)
- **Goals**: \([0, S, n]\)
- **Inference rules**:

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

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**:

\[
\begin{align*}
\text{Shift} & \quad \frac{[\alpha \bullet, j]}{[\alpha \omega_{j+1} \bullet, j + 1]} \\
\text{Reduce} & \quad \frac{[\alpha \gamma \bullet, j]}{[\alpha B \bullet, j]} \quad B \rightarrow \gamma
\end{align*}
\]
Bottom-up deduction: example

Parsing: top-down schema

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

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

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

Inference rules:

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

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

Top-down deduction: example

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

\[
\begin{align*}
[\bullet NP, 0] & \quad \text{axiom} \\
[\bullet NP PP, 0] & \quad \text{predict } NP \rightarrow NP PP \\
[\bullet D N PP, 0] & \quad \text{predict } NP \rightarrow D N \\
[\bullet the N PP, 0] & \quad \text{predict } D \rightarrow the \\
[\bullet N PP, 1] & \quad \text{scan} \\
[\bullet cat PP, 1] & \quad \text{predict } N \rightarrow cat \\
[\bullet PP, 2] & \quad \text{scan} \\
[\bullet P NP, 2] & \quad \text{predict } PP \rightarrow P NP \\
[\bullet in NP, 2] & \quad \text{predict } P \rightarrow in \\
[\bullet NP, 3] & \quad \text{scan} \\
[\bullet D N, 3] & \quad \text{predict } NP \rightarrow D N \\
[\bullet the N, 3] & \quad \text{predict } D \rightarrow the \\
[\bullet N, 4] & \quad \text{scan} \\
[\bullet hat, 4] & \quad \text{predict } N \rightarrow hat \\
[\bullet, 5] & \quad \text{scan}
\end{align*}
\]
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 → NP VP
S → Aux NP VP
S → VP
VP → Verb
VP → Verb NP
NP → Det Nominal
NP → Proper-Noun
Nominal → Noun
Nominal → Noun Nominal
Nominal → Nominal PP
PP → Prep NP

Det → that | this | a
Noun → book | flight | meal
Verb → book | include | includes
Prep → from | to | on
Proper-Noun → Houston | TWA
Aux → does

An example derivation tree

```
S
  / \  
VP  NP
  /  /  
Verb Det Noun
  /  /  
book that flight
```
**An example derivation tree**

```
S
  | NP
  | Nominal
  | PP
  | NP
  | Nominal
  | Det
  | Noun
  | Prep
  | Proper-Noun
  | Verb
  | Det
  | Noun
  | a
  | meal
```

```
S
  | NP
  | Nominal
  | PP
  | NP
  | Nominal
  | Aux
  | Det
  | Noun
  | Prep
  | Proper-Noun
  | Verb
  | Det
  | Noun
```

---

**Top-down vs. Bottom-up parsing**

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

To reduce “blind” search, add bottom-up filtering.

Observation: when trying to \(\text{Parse}(\beta, j)\), 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

```
Parse(β, j) ::
  if β = w_{j+1} · β' then return parse(β', j + 1)
  else if β = B · β' then
    if w_{j+1} is a left-corner of B 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

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

```
Det/Noun
  a/flight/from
  from/Chicago/to
  to/Houston/on
  on/TWA
```

Top-down parsing: repeated generation of sub-trees

```
Det/Noun
  a/flight/from
  from/Chicago/to
  to/Houston/on
  on/TWA
```

```
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

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

Basic concepts:

**Dotted rules:** if \( A \to \alpha \beta \) is a grammar rule then \( A \to \alpha \bullet \beta \) is a dotted rule.

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

- 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

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

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

**combination**:

$[i, A \rightarrow \alpha \bullet B\beta, k] \circ [k, B \rightarrow \gamma\bullet, j] = [i, A \rightarrow \alpha B\bullet \beta, j]$

Parsing: Earley deduction

**Inference rules**:

**Scan**

\[
[i, A \rightarrow \alpha \bullet w_{j+1}\beta, j] \quad \text{for } j := 1 \text{ to } n \quad \text{do}
\]

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

**Predict**

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

\[
[j, B \rightarrow \bullet \gamma, j]
\]

**Complete**

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

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

Early's parsing algorithm

**Parse ::**

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

\[
\text{for } j := 1 \text{ to } n \text{ do}
\]

\[
\text{for every rule } A \rightarrow w_j \text{ do}
\]

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

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

```plaintext
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])
```