Problems with DFTD Parser

- Left Recursion
- Ambiguity
- Inefficiency

### Left Recursion

- A grammar is left recursive if it contains at least one non-terminal A for which $A \Rightarrow \alpha A \beta$ and $\alpha \Rightarrow \epsilon$
- Intuitive idea: derivation of that category includes itself along its leftmost branch.

<table>
<thead>
<tr>
<th>NP $\Rightarrow$ NP and NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP $\Rightarrow$ Det Nominal</td>
</tr>
<tr>
<td>Det $\Rightarrow$ NP ' s</td>
</tr>
</tbody>
</table>

### Infinite Search

<table>
<thead>
<tr>
<th>S $\Rightarrow$ S</th>
</tr>
</thead>
<tbody>
<tr>
<td>S $\Rightarrow$ NP</td>
</tr>
<tr>
<td>VP $\Rightarrow$ VP</td>
</tr>
<tr>
<td>NP $\Rightarrow$ PP</td>
</tr>
<tr>
<td>PP $\Rightarrow$ PP</td>
</tr>
</tbody>
</table>

### Dealing with Left Recursion

- Reformulate the grammar
  - $A \Rightarrow A \beta | \alpha$
  - $A \Rightarrow \alpha A'$
  - $A' \Rightarrow \beta A' | \epsilon$
- Disadvantage: different (and probably unnatural) parse trees.
- Use a different parse algorithm

### Ambiguity

- Attachment Ambiguity: a constituent can be added to the parse tree in different places: *I shot an elephant in my pyjamas*
- Coordination Ambiguity: different scope of conjunction: *Black cats and dogs like to play*
Handling Disambiguation

- Statistical disambiguation
- Semantic knowledge.

Repeated Parsing of Subtrees

<table>
<thead>
<tr>
<th>Parse</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A flight from Indianapolis to Houston on TWA</td>
<td>3</td>
</tr>
</tbody>
</table>

Catalan Numbers

The $n$th Catalan number counts the ways of dissecting a polygon with $n+2$ sides into triangles by drawing nonintersecting diagonals.

<table>
<thead>
<tr>
<th>$n$</th>
<th># parses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>132</td>
</tr>
<tr>
<td>6</td>
<td>469</td>
</tr>
<tr>
<td>7</td>
<td>1430</td>
</tr>
<tr>
<td>8</td>
<td>4867</td>
</tr>
</tbody>
</table>

Earley Algorithm

- Dynamic Programming: solution involves filling in table of solutions to subproblems.
- Parallel Top Down Search
- Worst case complexity = $O(N^3)$ in length of sentence.
- Table, called a chart, contains $N+1$ entries

The Chart

- Each table entry contains a list of states
- Each state represents all partial parses that have been reached so far at that point in the sentence.
- States are represented using dotted rules containing information about:
  - Rule/subtree: which rule has been used
  - Progress: dot indicates how much of rule's RHS has been recognised.
  - Position: text segment to which this parse applies

Examples of Dotted Rules

- Initial S Rule
  $$S \rightarrow \bullet VP, [0, 0]$$
- Partially recognised NP
  $$NP \rightarrow \text{Det} \bullet \text{Nominal}, [1, 2]$$
- Fully recognised VP
  $$VP \rightarrow \text{V} \ VP \bullet, [0, 0]$$
- These states can also be represented graphically
The Chart

VP → V NP.

NP → Det Nominal

S → .VP

Earley Algorithm

- Main Algorithm: proceeds through each text position, applying one of the three operators below.
- **Predictor**: Creates "initial states" (ie states whose RHS is completely unparsed).
- **Scanner**: checks current input when next category to be recognised is pre-terminal.
- **Completer**: when a state is "complete" (nothing after dot), advance all states to the left that are looking for the associated category.

---

**Early Algorithm – Main Function**

```plaintext
function EARLEY-PARSE(words, grammar) returns chart

ENQUEUE((γ/S[0,0]), chart[0])

for i from 0 to LENGTH(words) do

for each state in chart[i] do

if INCOMPLETE?(state) and NEXT-CAT(state) is not a part of speech then

PREDICTOR(state)

else if INCOMPLETE?(state) and NEXT-CAT(state) is a part of speech then

SCANNER(state)

else

COMPLETER(state)

end

end

end

return chart
```

---

**Early Algorithm – Sub Functions**

```plaintext
procedure PREDICTOR((A/α/B[0,i], j))

for each (B/γ) in GRAMMAR-RULES-FOR(B, grammar) do

ENQUEUE((B/γ/B[0,j]), chart[j])

end

procedure SCANNER((A/α/B[0,i], j))

if B/PARTS-OF-SPEECH(word[j]) then

ENQUEUE(B → word[j]/γ/|j+, |j+1), chart[j+1])

end

procedure COMPLETER((B/γ/|j), k)

for each (A/α/B/β) in chart[j] do

ENQUEUE(A → A/α/B/β/γ/|j, chart[k])

end

end

procedure ENQUEUE(state, chart-entry)

if state is not already in chart-entry then

PUSH(state, chart-entry)

end

end
```

---

**Chart[0]**

<table>
<thead>
<tr>
<th>γ → S</th>
<th>[0,0] Dummy start state</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td>[0,0] Predictor</td>
</tr>
<tr>
<td>NP → Det NOMINAL</td>
<td>[0,0] Predictor</td>
</tr>
<tr>
<td>NP → Proper-Noun</td>
<td>[0,0] Predictor</td>
</tr>
<tr>
<td>S → Aux NP VP</td>
<td>[0,0] Predictor</td>
</tr>
<tr>
<td>S → Verb</td>
<td>[0,0] Predictor</td>
</tr>
<tr>
<td>VP → Verb</td>
<td>[0,0] Predictor</td>
</tr>
<tr>
<td>VP → Verb NP</td>
<td>[0,0] Predictor</td>
</tr>
</tbody>
</table>

---

**Chart[1]**

| Verb → book | [0,1] Scanner |
| VP → Verb | [0,1] Completer |
| S → VP | [0,1] Completer |
| VP → Verb NP | [0,1] Completer |
| NP → Det NOMINAL | [1,1] Predictor |
| NP → Proper-Noun | [1,1] Predictor |

---

**Chart[2]**

| Det → that | [1,2] Scanner |
| NP → Det NOMINAL | [1,2] Completer |
| NOMINAL → Noun | [2,2] Predictor |
| NOMINAL → Noun NOMINAL | [2,2] Predictor |
Retrieving Trees

- To turn recogniser into a parser, representation of each state must also include information about completed states that generated its constituents.

```
Retrieving Trees

• To turn recogniser into a parser, representation of each state must also include information about completed states that generated its constituents.

S18 Noun → flight [2,3] [ ] Scanner
S19 NOMINAL → Noun [2,3] [S18] Completer
S20 NOMINAL → Noun NOMINAL [2,3] [S18] Completer
S21 NP → Det NOMINAL [1,3] [S14,S19] Completer
S22 VP → Verb NP [0,3] [S8,S21] Completer
S23 S → VP [0,3] [S22] Completer
S24 NOMINAL → Noun [3,3] [ ] Predictor
S25 NOMINAL → Noun NOMINAL [3,3] [ ] Predictor
```

Extra Field