Parsing with CFGs

L445 / L545 / B659
Dept. of Linguistics, Indiana University
Spring 2016

Parsing with CFGs: Overview

Input: a string
Output: a (single) parse tree
▶ A useful step in the process of obtaining meaning
▶ We can view the problem as searching through all
possible parses (tree structures) to find the right one

Strategies:
▶ top-down (goal-directed) vs. bottom-up (data-directed)
▶ depth-first vs. breadth-first
▶ left-corner parsing (adding bottom-up to top-down)
▶ chart parsing (saving partial results)

Example grammar and desired tree

Sentence: Book that flight.

S → NP VP
S → Aux NP VP
S → VP
NP → Det Nominal
Nominal → Noun
Nominal → Nominal PP
NP → Proper-Noun
VP → Verb
VP → Verb NP

Direction of processing I
Top-down

Goal-driven processing is top-down:
▶ Start with the start symbol
▶ Derive sentential forms
   ▶ If the string is among the sentences derived this way, it
     is part of the language

Problem: Left-recursive rules (e.g., NP → NP PP) can give
rise to infinite hypotheses
▶ Plus, we can expand non-terms which cannot lead
to the existing input
▶ No tree takes the properties of the lexical items into
account until the last stage

How are alternatives explored?
I. Depth-first

At every choice point: Pursue a single alternative completely
before trying another alternative
▶ State of affairs at the choice points needs to be
remembered. Choices can be discarded after
unsuccessful exploration.
▶ Depth-first search is not necessarily complete.
Problem for top-down, left-to-right, depth-first processing:
▶ left-recursion
   For example, a rule like N' → N' PP leads to
   non-termination.
How are alternatives explored?

II. Breadth-first

At every choice point: Pursue every alternative for one step at a time

- Requires serious bookkeeping since each alternative computation needs to be remembered at the same time.
- Search is guaranteed to be complete.

Top-down, left-right, depth-first tree traversal

A walk-through

A walk-through (cont.)

There are still some choices that have to be made:

1. Which leaf node should be expanded first?
   - Left-to-right strategy moves through the leaf nodes in a left-to-right fashion

2. Which rule should be applied first for multiple rules with same LHS?
   - Can just use the textual order of rules from the grammar
   - There may be reasons to take rules in a particular order (e.g., probabilities)

Remaining choices

Syntactic rules:

\[
S \rightarrow NP \ VP \\
NP \rightarrow Det \ N \\
N \rightarrow dragon \\
\]

Lexicon:

\[
Vt \rightarrow saw \\
Det \rightarrow the \\
\]

Goal

Input

Action

<table>
<thead>
<tr>
<th>Goal</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>N VP</td>
<td>the young boy saw the dragon</td>
<td>expand S</td>
</tr>
<tr>
<td>Det N</td>
<td>young boy saw the dragon</td>
<td>expand NP</td>
</tr>
<tr>
<td>N</td>
<td>boy saw the dragon</td>
<td>consume the</td>
</tr>
<tr>
<td>N</td>
<td>saw the dragon</td>
<td>expand Vt</td>
</tr>
<tr>
<td>N</td>
<td>dragon</td>
<td>consume saw</td>
</tr>
<tr>
<td>N</td>
<td>VP</td>
<td>expand Det</td>
</tr>
<tr>
<td>Adj N</td>
<td>boy</td>
<td>consume the</td>
</tr>
<tr>
<td>Adj N</td>
<td>dragon</td>
<td>expand N</td>
</tr>
<tr>
<td>Adj N</td>
<td>the</td>
<td>consume dragon</td>
</tr>
<tr>
<td>Det</td>
<td>the</td>
<td>SUCCESS!</td>
</tr>
<tr>
<td>Det</td>
<td>a</td>
<td>SUCCESS!</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>SUCCESS!</td>
</tr>
<tr>
<td>VP</td>
<td>N</td>
<td>SUCCESS!</td>
</tr>
<tr>
<td>VP</td>
<td>the</td>
<td>SUCCESS!</td>
</tr>
<tr>
<td>N</td>
<td>dragon</td>
<td>SUCCESS!</td>
</tr>
</tbody>
</table>
Parsing with an agenda

Search states are kept in an agenda
- Search states consist of partial trees and a pointer to the next input word in the sentence

Based on what we’ve seen before, apply the next item on the agenda to the current tree
- Add new items to the agenda, based on the rules in the grammar which can expand at the (leftmost) node
  - We maintain the depth-first strategy by adding new hypotheses (rules) to the front of the agenda
- If we added them to the back, we would have a breadth-first strategy

A walk-through

Analysis | Input | Action
--- | --- | ---
NP saw | the dragon | reduce saw to Vt
NP Vt | the dragon | reduce the to Det
NP Vt the | dragon | reduce dragon to N
NP Vt Det | dragon | reduce Det N to NP
NP Vt Det dragon | <empty> | reduce Det N to NP
NP Vt Det N | <empty> | reduce Vt NP to VP
NP Vt NP | <empty> | reduce NP VP to S
S | <empty> | SUCCESS!
Grammar with left-corners

<table>
<thead>
<tr>
<th>Lexicon:</th>
<th>Syntactic rules:</th>
<th>Left corners:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vt → saw</td>
<td>S → NP VP</td>
<td>S ⇒ Det</td>
</tr>
<tr>
<td>Det → the</td>
<td>VP → Vt NP</td>
<td>VP ⇒ Vt</td>
</tr>
<tr>
<td>Det → a</td>
<td>NP → Det N</td>
<td>NP ⇒ Det</td>
</tr>
<tr>
<td>N → dragon</td>
<td>N → Adj N</td>
<td>N ⇒ Adj</td>
</tr>
<tr>
<td>Adj → young</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chart parsing

Problem: Inefficiency of recomputing subresults

Two example sentences and their potential analysis:

1. He [gave [the young cat] [to Bill]].
2. He [gave [the young cat] [some milk]].

The corresponding grammar rules:
- VP → V_{drans} NP PP<sub>to</sub>
- VP → V_{drans} NP NP

Regardless of final sentence analysis, the object NP (the young cat) will have the same analysis
- No need to analyze it twice

Cocke Younger Kasami (CYK) Algorithm

- Grammar has to be in Chomsky Normal Form (CNF):
  - RHS with a single terminal: A → a
  - RHS with two non-terminals: A → BC
  - no ε rules (A → ε)
- A representation of the string showing positions and word indices:
  \[ w_1 \ w_2 \ w_3 \ w_4 \ w_5 \ w_6 \]
  For example:
  \[ \text{the} \ \text{young} \ \text{boy} \ \text{saw} \ \text{the} \ \text{dragon} \]

Well-formed substring table (passive chart)

- The well-formed substring table, henceforth (passive) chart, for a string of length \( n \) is an \( n \times n \) matrix.
- The field \((i,j)\) of the chart encodes the set of all categories of constituents that start at position \( i \) and end at position \( j \), i.e.
  \[ \text{chart}(i,j) = \{ A \mid A \Rightarrow w_{i+1} \ldots w_{j} \} \]
- The matrix is triangular since no constituent ends before it starts.

Left corner parsing example

Consider again book that flight, with these rules:
- S → NP VP
- S → Aux NP VP
- S → VP
- NP → Det Nom.
- VP → Vt NP
- VP → Verb NP
- Nom. → Noun
- VP → Vt
- VP → Verb
- Det → Det
- PropN → Proper-Noun

With an ambiguous word like book, left corners tell us the noun reading is ruled out—it cannot start an S
- Thus, no NP expansions are considered

Solution: Chart Parsing (Memoization)

- Store intermediate results:
  a) completely analyzed constituents: **well-formed substring table** or **(passive) chart**
  b) partial and complete analyses: **(active) chart**
- In other words, instead of recalculating that the young cat is an NP, we’ll store that information
  - Dynamic programming: never go backwards
  - All intermediate results need to be stored for completeness.
  - All possible solutions are explored in parallel.
**Coverage Represented in the Chart**

An input sentence with 6 words:

\[ \cdot_0 \, w_1 \cdot_1 \, w_2 \cdot_2 \, w_3 \cdot_3 \, w_4 \cdot_4 \, w_5 \cdot_5 \, w_6 \cdot_6 \]

Coverage represented in the chart:

<table>
<thead>
<tr>
<th>FROM:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-1</td>
<td>0-2</td>
<td>0-3</td>
<td>0-4</td>
<td>0-5</td>
<td>0-6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1-2</td>
<td>1-3</td>
<td>1-4</td>
<td>1-5</td>
<td>1-6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2-3</td>
<td>2-4</td>
<td>2-5</td>
<td>2-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3-4</td>
<td>3-5</td>
<td>3-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4-5</td>
<td>4-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example for Coverage Represented in Chart**

Example sentence:

\[ \cdot_0 \, \text{the} \cdot_1 \, \text{young} \cdot_2 \, \text{boy} \cdot_3 \, \text{saw} \cdot_4 \, \text{the} \cdot_5 \, \text{dragon} \cdot_6 \]

Coverage represented in chart:

<table>
<thead>
<tr>
<th>FROM:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-1</td>
<td>0-2</td>
<td>0-3</td>
<td>0-4</td>
<td>0-5</td>
<td>0-6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1-2</td>
<td>1-3</td>
<td>1-4</td>
<td>1-5</td>
<td>1-6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2-3</td>
<td>2-4</td>
<td>2-5</td>
<td>2-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3-4</td>
<td>3-5</td>
<td>3-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4-5</td>
<td>4-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Parsing with a Passive Chart**

- The CKY algorithm is used, which:
  - explores all analyses in parallel, &
  - in a bottom-up fashion, &
  - stores complete subresults
- This algorithm is used to:
  - add top-down guidance (only use rules derivable from start-symbol), but avoid left-recursion problem
  - store partial analyses

**An Example for a Filled-in Chart**

Input sentence:

\[ \cdot_0 \, \text{the} \cdot_1 \, \text{young} \cdot_2 \, \text{boy} \cdot_3 \, \text{saw} \cdot_4 \, \text{the} \cdot_5 \, \text{dragon} \cdot_6 \]

<table>
<thead>
<tr>
<th>FROM:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-1</td>
<td>0-2</td>
<td>0-3</td>
<td>0-4</td>
<td>0-5</td>
<td>0-6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1-2</td>
<td>1-3</td>
<td>1-4</td>
<td>1-5</td>
<td>1-6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2-3</td>
<td>2-4</td>
<td>2-5</td>
<td>2-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3-4</td>
<td>3-5</td>
<td>3-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4-5</td>
<td>4-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Filling in the Chart**

- We build all constituents that end at a certain point before we build constituents that end at a later point.

<table>
<thead>
<tr>
<th>FROM:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-1</td>
<td>0-2</td>
<td>0-3</td>
<td>0-4</td>
<td>0-5</td>
<td>0-6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1-2</td>
<td>1-3</td>
<td>1-4</td>
<td>1-5</td>
<td>1-6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2-3</td>
<td>2-4</td>
<td>2-5</td>
<td>2-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3-4</td>
<td>3-5</td>
<td>3-6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4-5</td>
<td>4-6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**lexical_chart_fill(j-1, j)**

- Idea: Lexical lookup. Fill the field \(j - 1, j\) in the chart with the preterminal category dominating word \(j\).
- Realized as:

\[
\text{chart}(j - 1, j) := \{ X \mid X \rightarrow \text{word}_j \in P \} \]
syntactic_chart_fill(i,j)

- Idea: Perform all reduction steps using syntactic rules s.t. the reduced symbol covers the string from i to j.
- Realized as: chart(i,j) = { A | A → BC ∈ P, i < k < j, B ∈ chart(i,k), C ∈ chart(k,j) }
- Explicit loops over every possible value of k and every context free rule:
  chart(i,j) := {}. for k := i + 1 to j − 1 for every A → BC ∈ P
  if B ∈ chart(i,k) and C ∈ chart(k,j) then
  chart(i,j) := chart(i,j) ∪ {A}.

How memoization helps

If we look back to the chart for the sentence the young boy saw the dragon:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[Det]</td>
<td></td>
<td>[NP]</td>
<td></td>
<td>[VP]</td>
<td>[S]</td>
</tr>
<tr>
<td>1</td>
<td>[Adj]</td>
<td>[N]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>[V,N]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>[V,N]</td>
<td></td>
<td></td>
<td>[VP]</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[Det]</td>
<td>[NP]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[N]</td>
<td></td>
</tr>
</tbody>
</table>

- At cell (3,6), a VP is built by combining the V at (3,4) with the NP at (4,6), based on the rule VP → V NP
- Regardless of further processing, that VP is never rebuilt

Representing active chart items

- well-formed substring entry: chart(i,j,A): from i to j there is a constituent of category A
- More elaborate data structure needed to store partial results:
  - rule considered + how far processing has succeeded
  - dotted rule:
    |A → α • β|
  - active chart entry:
    chart(i,j,state(A,β)) Note: α is not represented A (incompletely) goes from i to j and can be completed by finding β

The Complete CYK Algorithm

Input: start category S and input string
n := length(string)

for j := 1 to n
  chart(i−1,j) := {X | X → wordj ∈ P}
for i := j − 2 down to 0
  chart(i,j) := {}
  for k := i + 1 to j − 1
    for every A → BC ∈ P
      if B ∈ chart(i,k) and C ∈ chart(k,j) then
        chart(i,j) := chart(i,j) ∪ {A}

Output: if S ∈ chart(0,n) then accept else reject

From CYK to Earley

- CKY algorithm:
  - explores all analyses in parallel
  - bottom-up
  - stores complete subresults
- desiderata:
  - add top-down guidance (to only use rules derivable from start-symbol), but avoid left-recursion problem of top-down parsing
  - store partial analyses (useful for rules right-hand sides longer than 2)
- Idea: also store partial results, so that the chart contains
  - passive items: complete results
  - active items: partial results

Dotted rule examples

- A dotted rule represents a state in processing a rule.
- Each dotted rule is a hypothesis:

  We found a vp if we still find
  vp → \bullet v-ditr np pp-to vp → v-ditr np pp-to vp → v-ditr np pp-to
  a v-ditr, a np, and a pp-to a np and a pp-to a np and a pp-to nothing

  - The first three are active items (or active edges)
  - The last one is a passive item/edge
The three actions in Earley’s algorithm

In \( [A \rightarrow \alpha \gamma B \beta] \) we call \( B \) the active constituent.

- **Prediction**: Search all rules realizing the active constituent.
- **Scanning**: Scan over each word in the input string.
- **Completion**: Combine an active edge with each passive edge covering its active constituent.

**Success state**: \( o[\text{start } \rightarrow \text{s } \cdot n] \)

A closer look at the three actions

**Prediction**

- **Prediction**: for each \( [A \rightarrow \alpha \gamma B \beta] \) in chart
- for each \( B \rightarrow \gamma \) in rules
- add \( [B \rightarrow \gamma] \) to chart

Prediction is the task of saying what kinds of input we expect to see

- Add a rule to the chart saying that we have not seen \( \gamma \), but when we do, it will form a \( B \)
- The rule covers no input, so it goes from \( j \) to \( j \)

Such rules provide the top-down aspect of the algorithm

Scanning

- Let \( w_1 \ldots w_j \ldots w_n \) be the input string
- For each \( [A \rightarrow \alpha \gamma w_j \beta] \) in chart
- Add \( [A \rightarrow \alpha w_j \gamma \beta] \) to chart

Scanning reads in lexical items

- We add a dotted rule indicating that a word has been seen between \( j-1 \) and \( j \)
- Such a completed dotted rule can be used to complete other dotted rules

These rules provide the bottom-up component to the algorithm

Completion

**Completion (fundamental rule of chart parsing):**

- For each \( [A \rightarrow \alpha \gamma B \beta] \) and \( [B \rightarrow \gamma \gamma] \) in chart
- Add \( [A \rightarrow \alpha B \gamma \beta] \) to chart

Completion combines two rules in order to move the dot, i.e., indicate that something has been seen

- A rule covering \( B \) has been seen, so any rule \( A \) which refers to \( B \) in its RHS moves the dot
- Instead of scanning from \( i \) to \( k \), \( A \) now spans from \( i \) to \( j \), which is where \( B \) ended

Once the dot is moved, the rule will not be created again

Eliminating scanning

**Scanning**

- For each \( [A \rightarrow \alpha \gamma w_j \beta] \) in chart
- Add \( [A \rightarrow \alpha w_j \gamma \beta] \) to chart

**Completion**

- For each \( [A \rightarrow \alpha \gamma B \beta] \) and \( [B \rightarrow \gamma \gamma] \) in chart
- Add \( [A \rightarrow \alpha B \gamma \beta] \) to chart

**Observation**: Scanning = completion + words as passive edges. One can thus simplify scanning to adding a passive edge for each word:

- For each \( w_j \) in \( w_1 \ldots w_n \)
- Add \( j \cdot [w_j \rightarrow \gamma] \) to chart

**Success state**: \( o[\text{start } \rightarrow s \cdot n] \)

Earley’s algorithm without scanning

**General setup**

Apply prediction and completion to every item added to chart

**Start**

- Add \( o[\text{start } \rightarrow \cdot s \text{ s}] \) to chart
- For each \( w_j \) in \( w_1 \ldots w_n \)
- Add \( j \cdot [w_j \rightarrow \cdot] \) to chart

**Success state**: \( o[\text{start } \rightarrow s \cdot n] \)
A tiny example grammar

Lexicon:
- vp → left
- det → the
- n → boy
- n → girl

Syntactic rules:
- s → np vp
- np → det n

An example run

<table>
<thead>
<tr>
<th>Start</th>
<th>Predict from 1</th>
<th>Predict from 2</th>
<th>Predict from 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s → •0 s</td>
<td>s → •0 np vp</td>
<td>s → •0 det n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complete 4 with 5</th>
<th>Complete 3 with 6</th>
<th>Predict from 7</th>
<th>Predict from 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>the → •1</td>
<td>det → the •1</td>
<td>det → det •1</td>
<td>n → •1 boy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complete 8 with 10</th>
<th>Complete 7 with 11</th>
<th>Complete 2 with 12</th>
<th>Predict from 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>n → •1 girl</td>
<td>boy → •2</td>
<td>n → boy •2</td>
<td>np → det n •2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scan &quot;boy&quot;</th>
<th>Complete 14 with 15</th>
<th>Complete 13 with 16</th>
<th>Complete 1 with 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>vp → •2</td>
<td>left → •1</td>
<td>vp → left •1</td>
<td>s → np vp •1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complete 1 with 17</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
</tr>
<tr>
<td>s → start → •3</td>
</tr>
</tbody>
</table>

Improving the efficiency of lexical access

- In the setup just described:
  - Words are stored as passive items so that prediction is used for preterminal categories
  - Set of predicted words for a preterminal can be huge
- If each word in the grammar is introduced by a preterminal rule `cat → word`, one can add a **passive item for each preterminal category** which can dominate the word instead of for the word itself
- What needs to be done:
  - syntactically distinguish syntactic rules from rules with preterminals on the left-hand side, i.e. lexical entries.
  - modify scanning to take lexical entries into account

Earley parsing

The Earley algorithm is efficient, running in polynomial time.

- Technically, however, it is a recognizer, not a parser

To make it a parser, each state needs to be augmented with a pointer to the states that its rule covers
  - For example, VP points to state where V was completed and state where NP was completed
  - Also true of the CKY algorithm: pointers need to be added to make it a parser