Polynomial-time Parsing with Context Free Grammars

CS114 Lecture 13
March 11, 2016
Professor Meteer

Thanks for Jurafsky & Martin & Prof. Pustejovksy for slides
### Parsing

**Computational task:**

Given a set of grammar rules and a sentence, find a valid parse of the sentence (efficiently)

Naively, you could try all possible trees until you get to a parse tree that conforms to the grammar rules, that has “S” at the root, and that has the right words at the leaves.

But that takes exponential time in the number of words.
Aspects of parsing

- Running a grammar backwards to find possible structures for a sentence
- Parsing can be viewed as a search problem
- Parsing is a hidden data problem
- For the moment, we want to examine all structures for a string of words
- We can do this bottom-up or top-down
  - This distinction is independent of depth-first or breadth-first search – we can do either both ways
  - We search by building a search tree which is distinct from the parse tree
Human parsing

- Humans often do ambiguity maintenance
  
  *Have the police ... eaten their supper?*
  
  *come in and look around.*
  
  *taken out and shot.*

- But humans also commit early and are “garden pathed”:
  
  - *The man who hunts ducks out on weekends.*
  
  - *The cotton shirts are made from grows in Mississippi.*
  
  - *The horse raced past the barn fell.*
A phrase structure grammar

- $S \rightarrow NP \ VP$
  - $N \rightarrow cats$
- $VP \rightarrow V \ NP$
  - $N \rightarrow claws$
- $VP \rightarrow V \ NP \ PP$
  - $N \rightarrow people$
- $NP \rightarrow NP \ PP$
  - $N \rightarrow scratch$
- $NP \rightarrow N$
  - $V \rightarrow scratch$
- $NP \rightarrow e$
  - $P \rightarrow with$
- $NP \rightarrow N \ N$
- $PP \rightarrow P \ NP$

- By convention, $S$ is the start symbol, but in the PTB, we have an extra node at the top (ROOT, TOP)
Phrase structure grammars = context-free grammars

- $G = (T, N, S, R)$
  - $T$ is set of terminals
  - $N$ is set of nonterminals
    - For NLP, we usually distinguish out a set $P \subseteq N$ of preterminals, which always rewrite as terminals
    - $S$ is the start symbol (one of the nonterminals)
    - $R$ is rules/productions of the form $X \rightarrow \gamma$, where $X$ is a nonterminal and $\gamma$ is a sequence of terminals and nonterminals (possibly an empty sequence)

- A grammar $G$ generates a language $L$. 
Probabilistic or stochastic context-free grammars (PCFGs)

- $G = (T, N, S, R, P)$
  - $T$ is set of terminals
  - $N$ is set of nonterminals
    - For NLP, we usually distinguish out a set $P \subset N$ of preterminals, which always rewrite as terminals
    - $S$ is the start symbol (one of the nonterminals)
    - $R$ is rules/productions of the form $X \rightarrow \gamma$, where $X$ is a nonterminal and $\gamma$ is a sequence of terminals and nonterminals (possibly an empty sequence)
    - $P(R)$ gives the probability of each rule.

\[
\forall X \in N, \sum_{X \rightarrow \gamma \in R} P(X \rightarrow \gamma) = 1
\]

- A grammar $G$ generates a language model $L$. 
Soundness and completeness

- A parser is *sound* if every parse it returns is valid/correct.
- A parser *terminates* if it is guaranteed to not go off into an infinite loop.
- A parser is *complete* if for any given grammar and sentence, it is sound, produces every valid parse for that sentence, and terminates.
- (For many purposes, we settle for sound but incomplete parsers: e.g., probabilistic parsers that return a *k*-best list.)
PARSING

- Parsing is the process of recognizing and assigning STRUCTURE.

- Parsing a string with a CFG:
  - Finding a derivation of the string consistent with the grammar.
  - The derivation gives us a PARSE TREE.

Grammar:

- S → NP VP
- S → Aux NP VP
- S → VP
- NP → Det Nom
- Nom → Noun
- Nom → Noun Nom

Diagram:

```
  S
 / \  
 VP  NP
 /   /  
 Verb Det Nom
 /     /  
 Book that Noun
           /  
           flight
```
The main problem with parsing is the existence of choice points.

**Parsing Strategy**
- **Top down:**
  - Expectation Driven
  - Start with “S”
- **Bottom up:**
  - Data Driven
  - Start with words/categories

**Search Strategy**
- Determining the order alternatives are considered
  - Depth first
  - Breadth first
TOP-DOWN vs BOTTOM-UP

- **TOP-DOWN:**
  - Only search among grammatical answers
  - BUT: suggests hypotheses that may not be consistent with data
  - Problem: left-recursion

- **BOTTOM-UP:**
  - Only forms hypotheses consistent with data
  - BUT: may suggest hypotheses that make no sense globally
If it’s not possible to examine all alternatives in parallel, it’s necessary to make further decisions:

- Which node in the current search space to expand first (breadth-first or depth-first)
- Which of the applicable grammar rules to expand first
- Which leaf node in a parse tree to expand next (e.g., leftmost)
TOP-DOWN, DEPTH-FIRST, LEFT-TO-RIGHT
TOP-DOWN, DEPTH-FIRST, LEFT-TO-RIGHT (II)
TOP-DOWN, DEPTH-FIRST, LEFT-TO-RIGHT (III)
function TOP-DOWN-PARSE(input, grammar) returns a parse tree

agenda ← (Initial S tree, Beginning of input)
current-search-state ← POP(agenda)
loop
  if SUCCESSFUL-PARSE?(current-search-state) then
    return TREE(current-search-state)
  else
    if CAT(NODE-TO-EXPAND(current-search-state)) is a POS then
      if CAT(node-to-expand) ⊆ POS(CURRENT-INPUT(current-search-state)) then
        PUSH(APPLY-LEXICAL-RULE(current-search-state), agenda)
      else
        return reject
    else
      PUSH(APPLY-RULES(current-search-state, grammar), agenda)
    if agenda is empty then
      return reject
    else
      current-search-state ← NEXT(agenda)
  end
end
Problems with top-down parsing

- Left recursive rules

- A top-down parser will do badly if there are many different rules for the same LHS.
  - Consider if there are 600 rules for S, 599 of which start with NP, but one of which starts with V, and the sentence starts with V.

- Useless work: expands things that are possible top-down but not fit the input

- Top-down parsers do well if there is useful grammar-driven control: search is directed by the grammar

- Top-down is hopeless for rewriting parts of speech (preterminals) with words (terminals). In practice that is always done bottom-up as lexical lookup.

- Repeated work: anywhere there is common substructure
A LEFT-RECURSIVE grammar may cause a T-D, D-F, L-R parser to never return.

Examples of left-recursive rules:

- NP → NP PP
- S → S and S
- But also:
  - NP → Det Nom
  - Det → NP’s
Repeated work...

Cats scratch people with cats with claws
Bottom-up parsing

- Bottom-up parsing is data directed
- The initial goal list of a bottom-up parser is the string to be parsed. If a sequence in the goal list matches the RHS of a rule, then this sequence may be replaced by the LHS of the rule.
- Parsing is finished when the goal list contains just the start category.
- If the RHS of several rules match the goal list, then there is a choice of which rule to apply (search problem)
- Can use depth-first or breadth-first search, and goal ordering.
- The standard presentation is as *shift-reduce parsing*. 
Problems with bottom-up parsing

• Unable to deal with empty categories: termination problem, unless rewriting empties as constituents is somehow restricted (but then it's generally incomplete)

• Useless work: locally possible, but globally impossible.

• Inefficient when there is great lexical ambiguity (grammar-driven control might help here)

• Conversely, it is data-directed: it attempts to parse the words that are there.

• Repeated work: anywhere there is common substructure
For Now

- Assume...
  - You have all the words already in some buffer
  - The input is not POS tagged prior to parsing
  - We won’t worry about morphological analysis
  - All the words are known
  - These are all problematic in various ways, and would have to be addressed in real applications.
Top-Down Search

Since we’re trying to find trees rooted with an $S$ (Sentences), why not start with the rules that give us an $S$.

Then we can work our way down from there to the words.
Top Down Space
Bottom-Up Parsing

- Of course, we also want trees that cover the input words. So we might also start with trees that link up with the words in the right way.

- Then work your way up from there to larger and larger trees.

```
S → NP VP
VP → Verb NP
NP → Det Nom
Nom → Nom Noun
Nom → Noun
Noun → flight
Det → that
Verb → book
```
Top-Down and Bottom-Up

- **Top-down**
  - Only searches for trees that can be answers (i.e. S’s)
  - But also suggests trees that are not consistent with any of the words

- **Bottom-up**
  - Only forms trees consistent with the words
  - But suggests trees that make no sense globally
Control

- Of course, in both cases we left out how to keep track of the search space and how to make choices
  - Which node to try to expand next
  - Which grammar rule to use to expand a node

- One approach is called backtracking.
  - Make a choice, if it works out then fine
  - If not then back up and make a different choice

- Problems:
  - Even with the best filtering, backtracking methods are doomed because of two inter-related problems
    - Ambiguity and search control (choice)
    - Shared subproblems
Ambiguity
Shared Sub-Problems

- No matter what kind of search (top-down or bottom-up or mixed) that we choose...
  - We can’t afford to redo work we’ve already done.
  - Without some help naïve backtracking will lead to such duplicated work.
Shared Sub-Problems

Consider

A flight from Indianapolis to Houston on TWA
Sample L1 Grammar

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Lexicon</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow NP \ VP$</td>
<td>$Det \rightarrow that \</td>
</tr>
<tr>
<td>$S \rightarrow Aux \ NP \ VP$</td>
<td>$Noun \rightarrow book \</td>
</tr>
<tr>
<td>$S \rightarrow VP$</td>
<td>$Verb \rightarrow book \</td>
</tr>
<tr>
<td>$NP \rightarrow Pronoun$</td>
<td>$Pronoun \rightarrow I \</td>
</tr>
<tr>
<td>$NP \rightarrow Proper-Noun$</td>
<td>$Proper-Noun \rightarrow Houston \</td>
</tr>
<tr>
<td>$NP \rightarrow Det \ Nominal$</td>
<td>$Aux \rightarrow does$</td>
</tr>
<tr>
<td>$Nominal \rightarrow Noun$</td>
<td>$Preposition \rightarrow from \</td>
</tr>
<tr>
<td>$Nominal \rightarrow Nominal \ Noun$</td>
<td></td>
</tr>
<tr>
<td>$Nominal \rightarrow Nominal \ PP$</td>
<td></td>
</tr>
<tr>
<td>$VP \rightarrow Verb$</td>
<td></td>
</tr>
<tr>
<td>$VP \rightarrow Verb \ NP$</td>
<td></td>
</tr>
<tr>
<td>$VP \rightarrow Verb \ NP \ PP$</td>
<td></td>
</tr>
<tr>
<td>$VP \rightarrow Verb \ PP$</td>
<td></td>
</tr>
<tr>
<td>$VP \rightarrow VP \ PP$</td>
<td></td>
</tr>
<tr>
<td>$PP \rightarrow Preposition \ NP$</td>
<td></td>
</tr>
</tbody>
</table>
Shared Sub-Problems

- Assume a top-down parse that has already expanded the *NP* rule (dealing with the Det)

- Now its making choices among the various *Nominal* rules

- In particular, between these two
  - *Nominal* -> *Noun*
  - *Nominal* -> *Nominal PP*

- Statically choosing the rules in this order leads to the following bad behavior...
Shared Sub-Problems
+ Shared Sub-Problems
Shared Sub-Problems
Shared Sub-Problems

```
*  
  NP
    Det  
      a
    Nominal
        Nominal
            Nominal
                Noun
                    flight
                PP
                    from Indianapolis
            PP
                on TWA
        PP
            to Houston
```
Dynamic Programming

- DP search methods fill tables with partial results and thereby
  - Avoid doing avoidable repeated work
  - Solve exponential problems in polynomial time (well not really)
  - Efficiently store ambiguous structures with shared sub-parts.

- We’ll cover two approaches that roughly correspond to top-down and bottom-up approaches.
  - CKY
  - Earley
The CKY (Cocke-Kasami-Younger) Algorithm

- Requires the grammar be in Chomsky Normal Form (CNF)
  - All rules must be in following form:
    - A → B C
    - A → w

- Any grammar can be converted automatically to Chomsky Normal Form
Chomsky Normal Form

- All rules are of the form
  \[ X \rightarrow Y Z \]
  \[ X \rightarrow w \]

- A transformation to this form doesn’t change the weak generative capacity of CFGs.
  - With some extra book-keeping in symbol names, you can even reconstruct the same trees with a detransform
  - Unaries/empties are removed recursively
  - N-ary rules introduce new nonterminals:
    - \[ VP \rightarrow V \ NP \ PP \] becomes \[ VP \rightarrow V \ @VP-V \] and \[ @VP-V \rightarrow NP \ PP \]

- In practice it’s a pain
  - Reconstructing n-aries is easy
  - Reconstructing unaries can be trickier

- But it makes parsing easier/more efficient
Converting to CNF

- Rules that mix terminals and non-terminals
  - Introduce a new dummy non-terminal that covers the terminal
    - INFVP -> to VP replaced by:
      - INFVP -> TO VP
      - TO -> to

- Rules that have a single non-terminal on right ("unit productions")
  - Rewrite each unit production with the RHS of their expansions

- Rules whose right hand side length >2
  - Introduce dummy non-terminals that spread the right-hand side
# Sample Grammar Conversion

<table>
<thead>
<tr>
<th>S → NP VP</th>
<th>S → NP VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → Aux NP VP</td>
<td>S → X1 VP</td>
</tr>
<tr>
<td>S → VP</td>
<td>S → book</td>
</tr>
<tr>
<td>S → V NP</td>
<td>S → VP PP</td>
</tr>
<tr>
<td>NP → Det Nom</td>
<td></td>
</tr>
<tr>
<td>NP → PrN</td>
<td></td>
</tr>
<tr>
<td>NP → Pronoun</td>
<td></td>
</tr>
<tr>
<td>Nom → Noun</td>
<td>Nom → Noun Nom</td>
</tr>
<tr>
<td>Nom → Noun Nom</td>
<td>Nom → Noun Nom</td>
</tr>
<tr>
<td>Nom → Nom PP</td>
<td>Nom → Nom PP</td>
</tr>
<tr>
<td>VP → V</td>
<td>VP → book</td>
</tr>
<tr>
<td>VP → V NP</td>
<td>VP → V NP</td>
</tr>
<tr>
<td>VP → VP PP</td>
<td>VP → VP PP</td>
</tr>
<tr>
<td>PP → Prep NP</td>
<td>PP → Prep NP</td>
</tr>
</tbody>
</table>
Given rules in CNF

Consider the rule $A \rightarrow BC$

- If there is an $A$ in the input then there must be a $B$ followed by a $C$ in the input.
- If the $A$ goes from $i$ to $j$ in the input then there must be some $k$ st. $i < k < j$
  - i.e. The $B$ splits from the $C$ someplace.

So let's build a table so that an $A$ spanning from $i$ to $j$ in the input is placed in cell $[i,j]$ in the table.

So a non-terminal spanning an entire string will sit in cell $[0, n]$

If we build the table bottom up we’ll know that the parts of the $A$ must go from $i$ to $k$ and from $k$ to $j$
Meaning that for a rule like \( A \rightarrow B \ C \) we should look for a \( B \) in \([i,k]\) and a \( C \) in \([k,j]\).

In other words, if we think there might be an \( A \) spanning \( i,j \) in the input... AND

\( A \rightarrow B \ C \) is a rule in the grammar THEN

There must be a \( B \) in \([i,k]\) and a \( C \) in \([k,j]\) for some \( i<k<j \)

So just loop over the possible \( k \) values
• Filling the \([i,j]^{th}\) cell in the CKY table
function CKY-PARSE(words, grammar) returns table

for j ← from 1 to LENGTH(words) do
    table[j − 1, j] ← \{ A \mid A \to words[j] \in grammar \}
for i ← from j − 2 downto 0 do
    for k ← i + 1 to j − 1 do
        table[i, j] ← table[i, j] \cup
        \{ A \mid A \to BC \in grammar,
        B \in table[i, k],
        C \in table[k, j] \}
We arranged the loops to fill the table a column at a time, from left to right, bottom to top.

This assures us that whenever we’re filling a cell, the parts needed to fill it are already in the table (to the left and below).

Are there other ways to fill the table?
<table>
<thead>
<tr>
<th>Book</th>
<th>the</th>
<th>flight</th>
<th>through</th>
<th>Houston</th>
</tr>
</thead>
<tbody>
<tr>
<td>S,VP,Verb Nominal, Noun</td>
<td>S,VP,X2</td>
<td>S, VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0,1]</td>
<td>[0,2]</td>
<td>[0,3]</td>
<td>[0,4]</td>
<td>[0,5]</td>
</tr>
<tr>
<td>Det</td>
<td>NP</td>
<td>NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1,2]</td>
<td>[1,3]</td>
<td>[1,4]</td>
<td>[1,5]</td>
<td></td>
</tr>
<tr>
<td>Nominal, Noun</td>
<td>Nominal, Noun</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2,3]</td>
<td>[2,4]</td>
<td>[2,5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prep</td>
<td>PP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3,4]</td>
<td>[3,5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP, Proper-Noun</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
0 Book 1 the 2 flight 3 through 4 Houston 5

**S → NP VP**

**S → X1 VP**

**X1 → AUX NP**

**S → Verb NP**

**S → VP PP**

Nom → book | flight | meal

Nom → Nom PP

Det → the | a | this

NP → Det Nom

NP → twa | houston

PP → Prep NP

Prep → through | in | at

VP → Verb

VP → Verb NP

Verb → book | fly | list
Example

John called Sue from Denver

John called Sue from Denver
Example 1: an alternative view

<table>
<thead>
<tr>
<th>S(0,5)</th>
<th></th>
<th></th>
<th></th>
<th>NP(4,5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P(3,4)</td>
<td>Denver</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NP(2,3)</td>
<td>from</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>V(2,3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>V(1,2)</td>
<td>Sue</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NP(0,1)</td>
<td>called</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>John</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example 1: Following the book**

<table>
<thead>
<tr>
<th>S -&gt; NP VP</th>
<th>NP(0,1)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VP -&gt; V NP</td>
<td>John</td>
<td>V(1,2)</td>
<td></td>
</tr>
<tr>
<td>VP -&gt; VP PP</td>
<td>called</td>
<td>NP(2,3)</td>
<td>V(2,3)</td>
</tr>
<tr>
<td>NP -&gt; NP PP</td>
<td>Sue</td>
<td>P(3,4)</td>
<td></td>
</tr>
<tr>
<td>PP -&gt; P NP</td>
<td>from</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP -&gt; John</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP -&gt; Sue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NP -&gt; Denver</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V -&gt; called</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V -&gt; sue</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P -&gt; from</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- S(0,5)
S -> NP VP
VP -> V NP
VP -> VP PP
NP -> NP PP
PP -> P NP
NP -> John
NP -> Sue
NP -> Denver
V -> called
V -> sue
P -> from

<table>
<thead>
<tr>
<th></th>
<th>NP(0,1)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>V(1,2)</td>
<td>VP(1,3)</td>
<td></td>
</tr>
<tr>
<td>called</td>
<td>NP(2,3)</td>
<td>V(2,3)</td>
<td>VP(1,5)</td>
</tr>
<tr>
<td></td>
<td>Sue</td>
<td>P(3,4)</td>
<td>PP(3,5)</td>
</tr>
<tr>
<td></td>
<td>from</td>
<td></td>
<td>NP(4,5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Denver</td>
</tr>
</tbody>
</table>
0 John 1 called 2 Sue 3 from 4 Denver 5

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>NP</td>
<td>VP</td>
<td>NP</td>
<td>VP</td>
<td>PP</td>
<td>PP</td>
</tr>
<tr>
<td>VP</td>
<td>V</td>
<td>NP</td>
<td>VP</td>
<td>PP</td>
<td>P</td>
<td>NP</td>
</tr>
<tr>
<td>NP</td>
<td>John</td>
<td>called</td>
<td>Sue</td>
<td>from</td>
<td>Denver</td>
<td></td>
</tr>
<tr>
<td>PP</td>
<td>P</td>
<td>VP</td>
<td>VP</td>
<td>P</td>
<td>NP</td>
<td>P</td>
</tr>
<tr>
<td>S -&gt;</td>
<td>NP</td>
<td>VP</td>
<td>NP</td>
<td>VP</td>
<td>PP</td>
<td>NP</td>
</tr>
<tr>
<td>VP -&gt;</td>
<td>V</td>
<td>NP</td>
<td>VP</td>
<td>PP</td>
<td>P</td>
<td>NP</td>
</tr>
<tr>
<td>VP -&gt;</td>
<td>VP</td>
<td>PP</td>
<td>VP</td>
<td>P</td>
<td>NP</td>
<td>P</td>
</tr>
<tr>
<td>NP -&gt;</td>
<td>NP</td>
<td>PP</td>
<td>NP</td>
<td>VP</td>
<td>PP</td>
<td>NP</td>
</tr>
<tr>
<td>NP -&gt;</td>
<td>John</td>
<td>called</td>
<td>Sue</td>
<td>from</td>
<td>Denver</td>
<td></td>
</tr>
<tr>
<td>NP -&gt;</td>
<td>Sue</td>
<td>called</td>
<td>Sue</td>
<td>from</td>
<td>Denver</td>
<td></td>
</tr>
<tr>
<td>NP -&gt;</td>
<td>Denver</td>
<td>called</td>
<td>Sue</td>
<td>from</td>
<td>Denver</td>
<td></td>
</tr>
<tr>
<td>V -&gt;</td>
<td>called</td>
<td>V</td>
<td>Sue</td>
<td>V</td>
<td>P</td>
<td>V</td>
</tr>
<tr>
<td>V -&gt;</td>
<td>sue</td>
<td>V</td>
<td>Sue</td>
<td>V</td>
<td>P</td>
<td>V</td>
</tr>
<tr>
<td>P -&gt;</td>
<td>from</td>
<td>P</td>
<td>Sue</td>
<td>V</td>
<td>P</td>
<td>V</td>
</tr>
<tr>
<td>P -&gt;</td>
<td>from</td>
<td>P</td>
<td>Sue</td>
<td>V</td>
<td>P</td>
<td>V</td>
</tr>
</tbody>
</table>

Production rules:
- S -> NP VP
- VP -> V NP
- VP -> VP PP
- NP -> NP PP
- PP -> P NP
- NP -> John
- NP -> Sue
- NP -> Denver
- V -> called
- V -> sue
- P -> from
John called Sue from Denver

| S -> NP VP | VP -> V NP | NP -> John | NP -> Sue | NP -> Denver |
| VP -> VP PP | PP -> P NP | NP -> NP PP | PP -> PP NP | PP -> PP NP |
| NP -> V NP | NP -> Sue | NP -> Denver | NP -> John | NP -> Denver |
| V -> V NP | V -> Sue | V -> Denver | V -> John | V -> John |

<table>
<thead>
<tr>
<th>0 John 1 called 2 Sue 3 from 4 Denver 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP(0,1)</td>
</tr>
<tr>
<td>John</td>
</tr>
<tr>
<td>called</td>
</tr>
<tr>
<td>Sue</td>
</tr>
<tr>
<td>from</td>
</tr>
<tr>
<td>Denver</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Back to Ambiguity

Did we solve it?

No…

- Both CKY and Earley will result in multiple S structures for the \([0,n]\) table entry.
- They both efficiently store the sub-parts that are shared between multiple parses.
- But neither can tell us which one is right.

Not a parser – a recognizer

- The presence of an S state with the right attributes in the right place indicates a successful recognition.
- But no parse tree… no parser
- That’s how we solve (not) an exponential problem in polynomial time
Converting CKY from Recognizer to Parser

- With the addition of a few pointers we have a parser
- Augment each new cell in chart to point to where we came from.
Problem (minor)

- We said CKY requires the grammar to be binary (ie. In Chomsky-Normal Form)

- We showed that any arbitrary CFG can be converted to Chomsky-Normal Form so that’s not a huge deal

- Except when you change the grammar the trees come out wrong

- All things being equal we’d prefer to leave the grammar alone.
Earley Parsing

- Allows arbitrary CFGs

- Where CKY is bottom-up, Earley is top-down

- Fills a table in a single sweep over the input words
  - Table is length N+1; N is number of words
  - Table entries represent
    - Completed constituents and their locations
    - In-progress constituents
    - Predicted constituents
**States**

- The table-entries are called states and are represented with dotted-rules.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S -&gt; · VP</td>
<td>A VP is predicted</td>
</tr>
<tr>
<td>NP -&gt; Det · Nominal</td>
<td>An NP is in progress</td>
</tr>
<tr>
<td>VP -&gt; V NP ·</td>
<td>A VP has been found</td>
</tr>
</tbody>
</table>

- It would be nice to know where these things are in the input so…

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S -&gt; · VP [0,0]</td>
<td>A VP is predicted at the start of the sentence</td>
</tr>
</tbody>
</table>
Graphically
As with most dynamic programming approaches, the answer is found by looking in the table in the right place.

In this case, there should be an S state in the final column that spans from 0 to n+1 and is complete.

If that’s the case you’re done.
  \[ S \rightarrow \alpha \cdot [0,n+1] \]
Earley Algorithm

- March through chart left-to-right.
- At each step, apply 1 of 3 operators
  - Predictor
    - Create new states representing top-down expectations
  - Scanner
    - Match word predictions (rule with word after dot) to words
  - Completer
    - When a state is complete, see what rules were looking for that completed constituent
Earley's example 1

Predict - Scan - Complete

PREDICT
S \rightarrow . NP VP \[0,0\]
NP \rightarrow . NP PP \[0,0\]
NP \rightarrow . John \[0,0\]
NP \rightarrow . Sue \[0,0\]
NP \rightarrow . Denver \[0,0\]

SCAN

COMPLETE

NP \rightarrow . John \[0,1\]
S \rightarrow NP . VP \[0,1\]
NP \rightarrow NP . PP \[0,1\]

Rules not predicted
P \rightarrow . V NP
VP \rightarrow . VP PP
PP \rightarrow . P NP
V \rightarrow . called
V \rightarrow . sue
P \rightarrow . from
+ Earley’s example step 2

**PREDICT**
S -> NP . VP [1,1]
NP -> NP . PP
VP -> . V NP
VP -> . VP PP
PP -> . P NP
V -> . Called
V -> . sue
P -> . from

**SCAN**

**COMPLETE**
V -> . called [1,2]
VP -> V . NP [2,2]

John called Sue from Denver
Earley’s example step 3

John called Sue from Denver

**PREDICT**
- S -> NP . VP
- NP -> NP . PP
- VP -> V . NP [2,2]
- VP -> VP PP
- PP -> P NP
- NP -> . John
- NP -> . Sue
- NP -> . Denver

**SCAN**
- NP -> . Sue

**COMPLETE**
- NP -> Sue . [2,3]
- VP -> V NP . [1,3]
- VP -> VP . PP [1,3]
- S -> NP VP . [0,3]
Earley’s example step 4

John called Sue from Denver

S -> NP . VP
NP -> NP . PP
VP -> V . NP
VP -> VP . PP
PP -> . P NP
P -> . from

NP -> . John
NP -> . Sue
NP -> . Denver

P -> . from

NP -> Denver
PP -> P NP .
NP -> NP PP .
VP -> VP PP .
VP -> V NP .
S -> NP VP .

DONE
+ Predictor

- Given a state
  - With a non-terminal to right of dot
    - That is not a part-of-speech category
  - Create a new state for each expansion of the non-terminal
  - Place these new states into same chart entry as generated state, beginning and ending where generating state ends.
- So predictor looking at
  - $S \rightarrow . \ VP [0,0]$
  - results in
    - $VP \rightarrow . \ Verb [0,0]$
    - $VP \rightarrow . \ Verb \ NP [0,0]$
Scanner

- Given a state
  - With a non-terminal to right of dot
    - That is a part-of-speech category
  - If the next word in the input matches this part-of-speech
    - Create a new state with dot moved over the non-terminal
- So scanner looking at
  - VP -> . Verb NP [0,0]
- If the next word, “book”, can be a verb, add new state:
  - VP -> Verb . NP [0,1]
- Add this state to chart entry following current one

- Note: Earley algorithm uses top-down input to disambiguate POS! Only POS predicted by some state can get added to chart!
Completer

- Applied to a state when its dot has reached right end of role.
  - *Means Parser has discovered a category over some span of input.*

- Find and advance all previous states that were looking for this category
  - copy state, move dot, insert in current chart entry

**Given:**
- NP -> Det Nominal . [1,3]
- VP -> Verb. NP [0,1]

**Add**
- VP -> Verb NP . [0,3]
Earley: how do we know we are done?

- How do we know when we are done?
- Find an S state in the final column that spans from 0 to n+1 and is complete.
- If that’s the case you’re done.
  - S → α · [0,n+1]
So sweep through the table from 0 to n+1…

- New predicted states are created by starting top-down from S
- New incomplete states are created by advancing existing states as new constituents are discovered
- New complete states are created in the same way.

More specifically…

1. Predict all the states you can upfront
2. Read a word
   1. Extend states based on matches
   2. Add new predictions
   3. Go to 2
3. Look at N+1 to see if you have a winner
**Example**

- Book that flight
- We should find… an S from 0 to 3 that is a completed state…
### Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
<th>[0,0]</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>$\gamma \rightarrow \bullet S$</td>
<td>Dummy state</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>$S \rightarrow \bullet NP \ VP$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>$S \rightarrow \bullet Aux \ NP \ VP$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>$S \rightarrow \bullet VP$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td>$NP \rightarrow \bullet Pronoun$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td>$NP \rightarrow \bullet Proper-Noun$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S6</td>
<td>$NP \rightarrow \bullet Det \ Nominal$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S7</td>
<td>$VP \rightarrow \bullet Verb$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S8</td>
<td>$VP \rightarrow \bullet Verb \ NP$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td>$VP \rightarrow \bullet Verb \ NP \ PP$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S10</td>
<td>$VP \rightarrow \bullet Verb \ PP$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>S11</td>
<td>$VP \rightarrow \bullet VP \ PP$</td>
<td>Pre-</td>
<td></td>
</tr>
<tr>
<td>Chart[1]</td>
<td>Rule</td>
<td>Production</td>
<td>[0,1]</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>------------</td>
<td>-------</td>
</tr>
<tr>
<td>S12</td>
<td>Verb</td>
<td>book *</td>
<td>[0,1]</td>
</tr>
<tr>
<td>S13</td>
<td>VP</td>
<td>Verb *</td>
<td>[0,1]</td>
</tr>
<tr>
<td>S14</td>
<td>VP</td>
<td>Verb * NP</td>
<td>[0,1]</td>
</tr>
<tr>
<td>S15</td>
<td>VP</td>
<td>Verb * NP PP</td>
<td>[0,0]</td>
</tr>
<tr>
<td>S16</td>
<td>VP</td>
<td>Verb * PP</td>
<td>[0,0]</td>
</tr>
<tr>
<td>S17</td>
<td>S</td>
<td>VP *</td>
<td>[0,1]</td>
</tr>
<tr>
<td>S18</td>
<td>VP</td>
<td>VP * PP</td>
<td>[0,1]</td>
</tr>
<tr>
<td>S19</td>
<td>NP</td>
<td>* Pronoun</td>
<td>[1,1]</td>
</tr>
<tr>
<td>S20</td>
<td>NP</td>
<td>* Proper-Noun</td>
<td>[1,1]</td>
</tr>
<tr>
<td>S21</td>
<td>NP</td>
<td>* Det Nominal</td>
<td>[1,1]</td>
</tr>
<tr>
<td>S22</td>
<td>PP</td>
<td>* Prep NP</td>
<td>[1,1]</td>
</tr>
<tr>
<td>Chart[2]</td>
<td>Rule</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>S23 Det → that •</td>
<td>[1,2]</td>
<td>Scanner</td>
<td></td>
</tr>
<tr>
<td>S24 NP → Det • Nominal</td>
<td>[1,2]</td>
<td>Completer</td>
<td></td>
</tr>
<tr>
<td>S25 Nominal → • Noun</td>
<td>[2,2]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S26 Nominal → • Nominal Noun</td>
<td>[2,2]</td>
<td>Predictor</td>
<td></td>
</tr>
<tr>
<td>S27 Nominal → • Nominal PP</td>
<td>[2,2]</td>
<td>Predictor</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chart[3]</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S28 Noun → flight •</td>
<td>[2,3]</td>
<td>Scanner</td>
</tr>
<tr>
<td>S29 Nominal → Noun •</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S30 NP → Det Nominal •</td>
<td>[1,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S31 Nominal → Nominal • Noun</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S32 Nominal → Nominal • PP</td>
<td>[2,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S33 VP → Verb NP •</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S34 VP → Verb NP • PP</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
<tr>
<td>S35 PP → • Prep NP</td>
<td>[3,3]</td>
<td>Predictor</td>
</tr>
<tr>
<td>S36 S → VP •</td>
<td>[0,3]</td>
<td>Completer</td>
</tr>
</tbody>
</table>
What kind of algorithms did we just describe (both Earley and CKY)

- Not parsers – recognizers
  - The presence of an S state with the right attributes in the right place indicates a successful recognition.
  - But no parse tree... no parser
  - That's how we solve (not) an exponential problem in polynomial time
Back to Ambiguity

- Did we solve it?
Ambiguity
Converting Earley from Recognizer to Parser

- With the addition of a few pointers we have a parser
- Augment the “Completer” to point to where we came from.
Augmenting the chart with structural information

<table>
<thead>
<tr>
<th>Step</th>
<th>Dotted rule</th>
<th>Span</th>
<th>Step</th>
<th>Backpointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>S8</td>
<td>Verb $\rightarrow$ book *</td>
<td>[0,1]</td>
<td>Scanner</td>
<td></td>
</tr>
<tr>
<td>S9</td>
<td>VP $\rightarrow$ Verb *</td>
<td>[0,1]</td>
<td>Completer</td>
<td>S8</td>
</tr>
<tr>
<td>S10</td>
<td>S $\rightarrow$ VP *</td>
<td>[0,1]</td>
<td>Completer</td>
<td>S9</td>
</tr>
<tr>
<td>S11</td>
<td>VP $\rightarrow$ Verb * NP</td>
<td>[0,1]</td>
<td>Completer</td>
<td>S8</td>
</tr>
<tr>
<td>S12</td>
<td>NP $\rightarrow$ * Det Nom</td>
<td>[1,1]</td>
<td>Predictor</td>
<td>S11</td>
</tr>
<tr>
<td>S13</td>
<td>NP $\rightarrow$ * PropN</td>
<td>[1,1]</td>
<td>Predictor</td>
<td>S11</td>
</tr>
</tbody>
</table>
Retrieving Parse Trees from Chart

- All the possible parses for an input are in the table.
- We just need to read off all the backpointers from every complete S in the last column of the table.
- Find all the S -> X . [0,N+1]
- Follow the structural traces from the Completer.
- Of course, this won’t be polynomial time, since there could be an exponential number of trees.
- So we can at least represent ambiguity efficiently.
How to do parse disambiguation

- Probabilistic methods
- Augment the grammar with probabilities
- Then modify the parser to keep only most probable parses
- And at the end, return the most probable parse