CSE 417
Dynamic Programming (pt 6)
Parsing Algorithms
Reminders

> HW9 due on Friday
  – start early
  – program will be slow, so debugging will be slow...
  – should run in 2-4 minutes

> Please fill out course evaluations
Dynamic Programming Review

Apply the steps...
1. Describe solution in terms of solution to any sub-problems
2. Determine all the sub-problems you’ll need to apply this recursively
3. Solve every sub-problem (once only) in an appropriate order

Key question:
1. Can you solve the problem by combining solutions from sub-problems?

Count sub-problems to determine running time
- total is number of sub-problems times time per sub-problem

optimal substructure: (small) set of solutions, constructed from solutions to sub-problems that is guaranteed to include the optimal one
Review From Previous Lectures

> Previously...

> Find opt substructure by considering how opt solution could use the last input
  - given multiple inputs, consider how opt uses last of either or both
  - given clever choice of sub-problems, find opt substructure by considering new options

> Alternatively, consider the shape of the opt solution in general: e.g., tree structured
Dynamic programming algorithms for parsing
   – CKY is an important algorithm and should be understandable
   – (everything after that is out of scope)

If you want to see more examples, my next two favorites are...
1. Optimal code generation (compilers)
2. System R query optimization (databases)
Outline for Today

- Grammars
- CKY Algorithm
- Earley’s Algorithm
- Leo Optimization
Grammars are used to understand languages

Important examples:
- natural languages
- programming languages
Natural Language Grammar

> Example:
Natural Language Grammar

> Input is a list of parts of speech
  - noun (N), verb (V), preposition (P), determiner (D), conjunction (C), etc.

Rachael Ray finds inspiration in cooking her family and her dog
Output is a tree showing structure

Rachael Ray finds inspiration in cooking her family and her dog
Programming Language Grammar

> Input is a list of “tokens”
  – identifiers, numbers, +, -, *, /, etc.
Programming Language Grammar

> Output is a tree showing structure
Programming Language Grammar

> Output is a tree showing structure
Definition: A context free grammar is a set of rules of the form

\[ A \rightarrow B_1 B_2 \ldots B_k \]

where each \( B_i \) can be either a token (a “terminal”) or another symbol appearing on the left-hand side of one of the rules (a “non-terminal”).

The output of parsing is a tree with leaves labeled by terminals, internal nodes labeled by non-terminals, and the children of internal nodes matching some rule from the grammar:

- e.g., can have a node labeled \( A \) with children \( B_1, B_2, \ldots, B_k \)
- want a specific non-terminal (“start” symbol) as the root
Context Free Grammars

Example grammar for only multiplication:

\[ F \rightarrow F \times N \]

\[ F \rightarrow N \]
Context Free Grammars

Example grammar for simple arithmetic expressions:

\[
\begin{align*}
F & \rightarrow F \ast N \\
F & \rightarrow N \\
T & \rightarrow T + F \\
T & \rightarrow F
\end{align*}
\]
Context Free Grammars

> Called “context free” because the rule $A \rightarrow B_1 B_2 ... B_k$ says that $A$ look like $B_1 B_2 ... B_k$ anywhere

> There are more general grammars called “context sensitive”
  - parsing those grammars is harder than NP-complete
  - (it is PSPACE-complete like generalized chess or go)
Context Free Grammars

> We will limit the sorts of grammars we consider...

> **Definition**: A grammar is in Chomsky normal form if *every* rule is in one of these forms:

1. \( A \rightarrow B \), where \( B \) is a terminal
2. \( A \rightarrow B_1 B_2 \), where both \( B_1 \) and \( B_2 \) are non-terminals

> In particular, this rules out empty rules: \( A \rightarrow \) – removal of those simplifies things *a lot*
**Context Free Grammars**

> **Definition:** A grammar is in Chomsky normal form if every rule is in one of these forms:

  1. $A \rightarrow C$, where $C$ is a terminal
  2. $A \rightarrow B_1 B_2$, where both $B_1$ and $B_2$ are non-terminals

> **Fact:** Any context free grammar can be rewritten into an equivalent one in Chomsky normal form

  – hence, we can assume this without loss of generality
  – (there can be some blowup in the size of the grammar though...)
Example grammar for arithmetic in Chomsky normal form
- step 1: remove terminals on right hand side

\[
\begin{align*}
F & \rightarrow F \ast N \\
F & \rightarrow N \\
T & \rightarrow T + F \\
T & \rightarrow F
\end{align*}
\]

\[
\begin{align*}
T & \rightarrow T + F \\
F & \rightarrow F \ast N \\
F & \rightarrow N
\end{align*}
\]
Context Free Grammars

> Example grammar for arithmetic in Chomsky normal form
  – step 1: remove terminals on right hand side

\[
\begin{align*}
F &\rightarrow F \ast N \\
F &\rightarrow N \\
T &\rightarrow T + F \\
T &\rightarrow F \\
M &\rightarrow * \\
F &\rightarrow F M N \\
F &\rightarrow N \\
P &\rightarrow + \\
\end{align*}
\]
Example grammar for arithmetic in Chomsky normal form
- step 2: introduce new non-terminals to replace 3+ on right hand side

\[
\begin{align*}
F &\rightarrow F \cdot N \\
F &\rightarrow N \\
T &\rightarrow T + F \\
T &\rightarrow F \\
M &\rightarrow \star \\
F &\rightarrow F \cdot M \cdot N \\
F &\rightarrow N \\
P &\rightarrow +
\end{align*}
\]
Context Free Grammars

Example grammar for arithmetic in Chomsky normal form
- step 2: introduce new non-terminals to replace 3+ on right hand side

\[
\begin{align*}
F & \rightarrow F \ast N \\
F & \rightarrow N \\
T & \rightarrow T + F \\
T & \rightarrow F \\
T & \rightarrow F_1 F \\
T_1 & \rightarrow T P \\
F & \rightarrow F_1 N \\
F_1 & \rightarrow F \ast M \\
F & \rightarrow N \\
M & \rightarrow \ast \\
P & \rightarrow +
\end{align*}
\]
Example grammar for arithmetic in Chomsky normal form
- step 3: eliminate 1 non-terminal on RHS by substitution

\[
\begin{align*}
    F &\rightarrow F \ast N \\
    F &\rightarrow N \\
    T &\rightarrow T + F \\
    T &\rightarrow F \\
    T_1 &\rightarrow T P \\
    M &\rightarrow * \\
    F &\rightarrow F_1 N \\
    F_1 &\rightarrow F M \\
    F &\rightarrow N \\
    P &\rightarrow +
\end{align*}
\]
Example grammar for arithmetic in Chomsky normal form
- step 3: eliminate 1 non-terminal on RHS by substitution

\[
\begin{align*}
F & \rightarrow F \ast N \\
F & \rightarrow N \\
T & \rightarrow T + F \\
T & \rightarrow F \\
T & \rightarrow T_1 F \\
T_1 & \rightarrow T P \\
T_1 & \rightarrow F P \\
T & \rightarrow F_1 N \\
T & \rightarrow F_1 N \\
M & \rightarrow * \\
P & \rightarrow +
\end{align*}
\]
Outline for Today

- Grammars
- CKY Algorithm
- Earley’s Algorithm
- Leo Optimization
Trying to find a tree...

Q: What technique do we know that might be helpful?
A: Dynamic programming!
Apply dynamic programming...
- to find any tree that matches the data
- (can be generalized to find the “most likely” parse also...)

Think about what the parse tree for tokens 1 .. n might look like
- root corresponds to some rule $A \rightarrow B_1 B_2$ (Chomsky Normal Form)
- child $B_1$ is root of parse tree for some 1 .. k
- child $B_2$ is root of parse tree for k+1 .. n
- (or it could be a leaf $A \rightarrow C$, where C is a terminal, if n=1)
In general, parse tree for tokens $i .. j$ might look like

- $A \to C$ if $i = j$ OR
- $A \to B_1 B_2$ where
  - child $B_1$ is root of parse tree for some $i .. k$
  - child $B_2$ is root of parse tree for $k+1 .. j$

Try each of those possibilities (at most $|G|$) for each $(i,j)$ pair

- each requires checking $j - i + 1$ possibilities for $k$
- need answers to sub-problem with $j - i$ smaller
  - can fill in the table along the diagonals, for example
Cocke–Kasami–Younger (CKY)

Example table from arithmetic example:

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>*</th>
<th>4</th>
<th>+</th>
<th>5</th>
<th>*</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>F/T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>F/T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>F/T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F/T</td>
<td></td>
</tr>
</tbody>
</table>
Cocke–Kasami–Younger (CKY)

Example table from arithmetic example:

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>*</th>
<th>4</th>
<th>+</th>
<th>5</th>
<th>*</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>F/T</td>
<td>F₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>F/T</td>
<td>T₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>F/T</td>
<td>F₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F/T</td>
<td></td>
</tr>
</tbody>
</table>
Cocke–Kasami–Younger (CKY)

> Example table from arithmetic example:

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>*</th>
<th>4</th>
<th>+</th>
<th>5</th>
<th>*</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>F/T</td>
<td>$F_1$</td>
<td>F/T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>F/T</td>
<td>$T_1$</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>F/T</td>
<td>$F_1$</td>
<td>F/T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F/T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T → $T_1$ F
T → F₁ N
$T_1$ → T P
$T_1$ → F P
F → F₁ N
F₁ → F M
T → N
F → N
M → *
P → +

W
**Cocke–Kasami–Younger (CKY)**

Example table from arithmetic example:

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>*</th>
<th>4</th>
<th>+</th>
<th>5</th>
<th>*</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>F/T</td>
<td>F₁</td>
<td>F/T</td>
<td></td>
<td>T₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>F/T</td>
<td></td>
<td></td>
<td>T₁</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>F/T</td>
<td>F₁</td>
<td>F/T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F/T</td>
<td></td>
</tr>
</tbody>
</table>

**Production rules:**

- T → T₁ F
- T → F₁ N
- T₁ → T P
- T₁ → F P
- F → F₁ N
- F₁ → F M
- T → N
- F → N
- M → *
- P → +
Cocke–Kasami–Younger (CKY)

> Example table from arithmetic example:

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>*</th>
<th>4</th>
<th>+</th>
<th>5</th>
<th>*</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>F/T</td>
<td>F</td>
<td>F/T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>F/T</td>
<td>T</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>F/T</td>
<td>F</td>
<td>F/T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F/T</td>
<td></td>
</tr>
</tbody>
</table>
## Cocke–Kasami–Younger (CKY)

> Example table from arithmetic example:

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>*</th>
<th>4</th>
<th>+</th>
<th>5</th>
<th>*</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>F/T</td>
<td>F₁</td>
<td>F/T</td>
<td>T₁</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F/T</td>
<td>T₁</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>F/T</td>
<td>F₁</td>
<td>F/T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F/T</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

T → T₁ F
T₁ → T P
T₁ → F P
F → F₁ N
F₁ → F M
T → N
F → N
M → *
P → +
Cocke–Kasami–Younger (CKY)

Can reconstruct the tree from the table as usual.

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>*</th>
<th>4</th>
<th>+</th>
<th>5</th>
<th>*</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>F/T</td>
<td>F₁</td>
<td>F/T</td>
<td>T₁</td>
<td>T</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F/T</td>
<td>T₁</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>F/T</td>
<td>F₁</td>
<td>F/T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>F/T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cocke–Kasami–Younger (CKY)

> Running time is $O(|G| \cdot n^3)$
  - in NLP, $|G| >> n$, so this is great
  - in PL, $|G| < n$, so this is not great
  - in algorithms, this is usually considered $O(n^3)$ since $|G|$ is a "constant"
    > I will follow this convention for the rest of the lecture...

> Algorithm easily generalizes to find “most likely” parse tree
  - frequently used in NLP case
Outline for Today

- Grammars
- CKY Algorithm
- Earley’s Algorithm
- Leo Optimization
Improving CKY (out of scope)

- CKY is not optimal even for general grammars...
  - (can be improved using fast matrix multiplication)

- PLUS we know that certain grammars can be parsed much faster
- In particular, there exist $O(n)$ algorithms for typical PL grammars
  - $O(n^3)$ was out of the question in 1965...

- Arithmetic example is one of those
  - notice how the table is mostly blank
  - that's a lot of wasted effort
Improving CKY

> To get to $O(n)$, we cannot fill in an $n \times n$ table
  – doing so always requires $\Omega(n^2)$ time
**Improving CKY**

> **Idea:** coalesce columns...

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>*</th>
<th>4</th>
<th>+</th>
<th>5</th>
<th>*</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>N/F/T</td>
<td>$F_1$</td>
<td>F/T</td>
<td>$T_1$</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>*</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>N/F/T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N/F/T</td>
<td>$F_1$</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N/F/T</td>
</tr>
</tbody>
</table>
Improving CKY

> **Idea:** coalesce columns...
  > - let $I_j$ include everything in the column $j$
  > - these are rules that parse $i \ldots j$ for some $i$

> Need to remember $i$ as well
  > - write entries of $I_j$ as “A (i)”, recording both symbol and where parsing started
Improving CKY

Now we fill in the sets $I_1, I_2, ..., I_n$ — parsing left to right

If we are lucky enough to get $|I_j| = O(1)$ for all $j$, this *could* be a linear time algorithm — assuming we can build $I_j$ in $O(1)$ time

Latter means we *cannot* look at all previous $I_j$'s — probably need to only look at $I_{j-1}$
Suppose $I_j$ is the set of “A (i)” where A matches i .. j

How do we build $I_j$?

If N is the j-th symbol of input, add “A (j)” for every $A \rightarrow N$ rule

What next?

If “C (k)” is in $I_j$, we might need to add “A (i)” for any $A \rightarrow B C$...

- “A (i)” should be added if “B (i)” is in $I_k$
- it takes $O(n)$ time to try every k in 1 .. j-1
- so we are back to $\Omega(n^2)$
To get O(n), we need to keep track of anything we might need to use later on in order to complete the parsing of a rule.

Specifically, if we have parsed “B (i)”, we need to keep track of the fact that it could be used to get an A → B C (i) if we later see C.

We write this fact as “A → B · C (i)”, which, in Ij, means that we have parsed the B part at i .. j.

- (the “C” part can be missing here i.e., if the rule is A → B, where B is a non-terminal)
Let \( I_j \) be the set of elements like “\( A \rightarrow B \cdot C (i) \)”, where:
1. \( B \) matches input tokens \( i \ldots j \)
2. It is possible for \( A \) to follow something that matches input tokens \( 1 \ldots i-1 \)

Note that “\( \cdot \)” can be at beginning, middle, or end
   - (we may as well drop the limit of only 2 symbols on the RHS)

Second part is another optimization
   - don’t waste time trying to parse rules that aren’t useful based on what came earlier
Let $I_j$ be the set of elements like “A $\rightarrow$ B $\cdot$ C (i)”, where:
1. B matches input tokens $i..j$
2. It is possible for A to follow something that matches input tokens $1..i-1$

Fill in $I_j$ as follows:
- add anything that could follow $I_{j-1}$ and matches input token j
  > (if “A $\rightarrow$ B $\cdot$ C (i)” is in $I_{j-1}$, then C could follow)
- for each added complete item “A $\rightarrow$ B C $\cdot$ (i)” added:
  > if $I_i$ contains “A’ $\rightarrow$ B $\cdot$ A (i’)”, then add “A’ $\rightarrow$ B A $\cdot$ (i)” to $I_j$
  > (likewise for “A’ $\rightarrow$ $\cdot$ A (i’)”)
- add all those items that could follow the ones already added
Improved Parser

> Fill in $I_j$ as follows:
  > – add anything that could follow $I_{j-1}$ and matches input token $j$
  >    > (if “$A \rightarrow B \cdot C (i)$” is in $I_{j-1}$, then $C$ could follow)
  > – for each added complete item “$A \rightarrow B \cdot C (i)$” added:
  >    > if $I_i$ contains “$A' \rightarrow B \cdot A (i')$”, then add “$A' \rightarrow B A \cdot (i')$” to $I_j$
  >    > (likewise for “$A' \rightarrow A (i')$”)
  > – add all those items that could follow the ones already added

> If all $|I_j|$’s are size $O(1)$, then this is $O(1)$ time per item
  > – hence, $O(n)$ over all
Earley’s algorithm

> This version is called Earley’s algorithm

> It was developed independently of CKY by Earley
  – (relation to CKY was noted by Ruzzo et al.)
  – also considered a dynamic programming algorithm
    > the sub-problems being solved are not quite so obvious as in CKY
Earley’s algorithm

> Can be shown that Earley’s algorithm runs in $O(n^2)$ time for any unambiguous grammar
  – meaning there is only one possible parse tree
    > typical of PL grammars (though not NLP grammars)

> Can also be shown it runs in $O(n)$ time for nice LR(k) grammars
> BUT not for all LR(k) grammars
  – latter can be parsed in $O(n)$ time by other algorithms

> The running time is at least the sum of sizes of the $I_j$’s...
Outline for Today

- Grammars
- CKY Algorithm
- Earley’s Algorithm
- Leo Optimization
Bad Cases for Earley

Q: Can the Ij’s be O(n) for some unambiguous grammar’s?
Bad Cases for Earley

> Q: Can the Ij’s be $O(n)$ for some unambiguous grammar’s?
> A: Unfortunately, yes

A $\rightarrow$ a
B $\rightarrow$ b
B $\rightarrow$ A B

> All B’s completed in $I_n$
> Q: Can the Ij’s be O(n) for some \textit{unambiguous} grammar’s?
> A: Unfortunately, yes

> This is a “right recursive grammar”
> Fortunately, these are the only bad cases (O(n) otherwise)

> Grammars can be \textit{usually} be rewritten to avoid it
Joop Leo’s Optimization

> Alternatively, we can improve the algorithm to handle those...

> Leo makes the following optimization:
  – only record the top-most item in a tall stack like this
  – (actually $O(1)$ copies of it depending on how we might look for it later)

> Can then show that the $I_j$’s are $O(1)$ size
  – number with dot not at end is $O(1)$ due to LR(k) property
  – clever argument shows number with dot at end is also $O(1)$
    > removing stacks leaves tree with all 2+ children and leaves those above
    > (furthermore, each is discovered only once for unambiguous grammars)
Alternatively, we can improve the algorithm to handle those...

Leo makes the following optimization:
- only record the top-most item in a tall stack like this
- (actually $O(1)$ copies of it depending on how we might look for it later)

Result is $O(n)$ in the worst case for LR(k)
- (i.e., for anything parsable by deterministic push-down automaton
- covers almost every PL grammar
CKY and Earley are used in NLP
- recall that \(|G|\) is usually larger there

In PL, we typically use special grammars (e.g., LR(k)) that can be parsed in linear time
- LR(k) was invented by Don Knuth
- parses anything that can be parsed by a deterministic push-down automaton

Earley + Leo gives the same asymptotic performance
- expect it to see more use given speed of computers
- (LR parsing was developed for machines 10k x slower)