Lecture 10: General and Bottom-Up Parsing

Job Opportunity. Professor Keltner of the Psychology Department is looking for a web developer to help with a moodle system (CMS). There are options for a stipend, and if the project is completed on schedule the developer’s work will be shown on a TEDx presentation. See Piazzza for more details.
A Little Notation

Here and in lectures to follow, we’ll often have to refer to general productions or derivations. In these, we’ll use various alphabets to mean various things:

- **Capital roman letters are nonterminals (A, B, ...).**
- **Lower-case roman letters are terminals (or tokens, characters, etc.)**
- **Lower-case greek letters are sequences of zero or more terminal and nonterminal symbols, such as appear in sentential forms or on the right sides of productions (α, β, ...).**
- **Subscripts on lower-case greek letters indicate individual symbols within them, so \( \alpha = \alpha_1\alpha_2 \ldots \alpha_n \) and each \( \alpha_i \) is a single terminal or nonterminal.**

For example,

- **A : α** might describe the production \( e : e ' + ' t, \)
- **B ⇒ αAγ ⇒ αβγ** might describe the derivation steps \( e ⇒ e ' + ' t \)
  \( ⇒ e ' + ' ID \) (α is e ’+’; A is t; B is e; and γ is empty.)
Fixing Recursive Descent

• First, let’s define an impractical but simple implementation of a top-down parsing routine.

• For nonterminal $A$ and string $S = c_1 c_2 \ldots c_n$, we’ll define $\text{parse}(A, S)$ to return the length of a valid substring derivable from $A$.

• That is, $\text{parse}(A, c_1 c_2 \ldots c_n) = k$, where

$$
\underbrace{c_1 c_2 \ldots c_k}_{A \rightarrow^*} c_{k+1} c_{k+2} \ldots c_n
$$
Abstract body of \text{parse}(A,S)

- Can formulate top-down parsing analogously to NFAs.

\text{parse} (A, S):

"""Assuming \( A \) is a nonterminal and \( S = c_1c_2\ldots c_n \) is a string, return integer \( k \) such that \( A \) can derive the prefix string \( c_1\ldots c_k \) of \( S \)."""

Choose production ‘\( A: \alpha_1\alpha_2\ldots\alpha_m \)’ for \( A \) (nondeterministically)

\( k = 0 \)

for \( x \) in \( \alpha_1, \alpha_2, \ldots, \alpha_m \):

if \( x \) is a terminal:

if \( x == c_{k+1} \):

\( k += 1 \)

else:

GIVE UP

else:

\( k += \text{parse} (x, c_{k+1}\ldots c_n) \)

return \( k \)

- Assume that the grammar contains one production for the start symbol: \( p: \gamma \Rightarrow \).

- We’ll say that a call to \text{parse} returns a value if some set of choices for productions (the blue step) would return a value (just like NFA).

- Then if \( \text{parse}(p, S) \) returns a value, \( S \) must be in the language.
Example

Consider parsing $S=\text{ID*ID-\textbackslash'}$ with a grammar from last time:

$$
p : e \ ' -' \\
e : t \\
\quad \mid e \ '/' t \\
\quad \mid e \ '*' t \\
t : \text{ID}
$$
Example

Consider parsing \( S=\text{"ID*ID-"} \) with a grammar from last time:

\[
\begin{align*}
p & : e \, \cdot \, \cdot \\
e & : t \\
& \mid e \, / \, t \\
& \mid e \, * \, t \\
t & : \text{ID}
\end{align*}
\]

\( k_i \) means “the variable \( k \) in the call to \( \text{parse} \) that is nested \( i \) deep.” Outermost \( k \) is \( k_1 \).

A failing path through the program:

\[
\begin{align*}
\text{parse}(p, S): & \\
& \text{Choose } p : e \, \cdot \, \cdot \\
\text{parse}(e, S): & \\
& \text{Choose } e : t: \\
\text{parse}(t, S): & \\
& \text{Choose } t : \text{ID}: \\
& \text{choose } t : \text{ID}: \\
& \text{check } S[1] == \text{ID}; \text{ OK, so } k_3 += 1; \\
& \text{return 1 } \quad ( = k_3; \text{ added to } k_2) \\
& \text{return 1 } \quad (\text{and add to } k_1) \\
\text{Check } S[2] == S[k_1+1] == \cdot \, \cdot & : \text{ GIVE UP } \quad (S[2] == \cdot \, \cdot)
\end{align*}
\]
Consider parsing $S=\text{"ID\text{*}\text{ID-\text{\text{"}}}}$ with a grammar from last time:

\[
\begin{align*}
p &: e \ '⊣' \\
e &: t \\
& \quad | e \ '/' t \\
& \quad | e \ '*' t \\
t &: \text{ID}
\end{align*}
\]

A successful path through the program:

- $\text{parse}(p, S)$:
  - Choose $p : e \ '⊣'$:
  - $\text{parse}(e, S)$:
    - Choose $e : e \ '*' t$:
      - $\text{parse}(e, S)$:
        - Choose $e : t$:
          - $\text{parse}(t, S)$:
            - Choose $t : \text{ID}$:
              - check $S[1] == \text{ID}$; OK, so return 1
              - return 1 (so $k_2 += 1$)
              - check $S[k_2] == \text{"\text{"}}$; OK, $k_2 += 1$
              - $\text{parse}(t, S_3)$: # $S_3 == \text{"ID \text{"}}$
                - Choose $t : \text{ID}$:
                  - check $S_3[k_3+1] == S_3[1] == \text{ID}$; OK
                  - $k_3+=1$; return 1 (so $k_2 += 1$)
                  - return 3
              - Check $S[k_1+1] == S[4] == \text{"\text{"\text{"}}}$: OK
              - $k_1 +=1$; return 4

$k_i$ means "the variable $k$ in the call to parse that is nested $i$ deep." Outermost $k$ is $k_1$. Likewise for $S$. 
Making a Deterministic Algorithm

- If we had an infinite supply of processors, could just spawn new ones at each “Choose” line.
- Some would give up, some loop forever, but on correct programs, at least one processor would get through.
- To do this for real (say with one processor), need to keep track of all possibilities systematically.
- This is the idea behind Earley’s algorithm:
  - Handles any context-free grammar.
  - Finds all parses of any string.
  - Can recognize or reject strings in $O(N^3)$ time for ambiguous grammars, $O(N^2)$ time for “nondeterministic grammars”, or $O(N)$ time for deterministic grammars (such as accepted by Bison).
Earley's Algorithm: I

- First, reformulate to use recursion instead of looping. Assume the string $S = c_1 \cdots c_n$ is fixed.

- Redefine `parse`:

  ```python
  parse (A: α • β, s, k):
  
  """Assumes A: αβ is a production in the grammar, 0 <= s <= k <= n, and α can produce the string $c_{s+1} \cdots c_k$. Returns integer j such that β can produce $c_{k+1} \cdots c_j$."
  ````

- Or diagrammatically, `parse` returns an integer $j$ such that:

\[
\begin{array}{c}
C_1 \cdots C_s \overbrace{C_{s+1} \cdots C_k} \overbrace{C_{k+1} \cdots C_j} C_{j+1} \cdots C_n \\
\alpha \Rightarrow \quad \beta \Rightarrow
\end{array}
\]
Earley's Algorithm: II

parse (A: α • β, s, k):

"""Assumes A: αβ is a production in the grammar, 0 <= s <= k <= n, and α can produce the string c_{s+1}\cdots c_k.
Returns integer j such that β can produce c_{k+1}\cdots c_j."""

if β is empty:
  return k

Assume β has the form xδ
if x is a terminal:
  if x == c_{k+1}:
    return parse(A: αx • δ, s, k+1)
  else:
    GIVE UP
else:
  Choose production ‘x: κ’ for x (nondeterministically)
  j = parse(x: •κ, k, k)
  return parse (A: αx • δ, s, j)

- Now do all possible choices that result in such a way as to avoid redundant work (“nondeterministic memoization”).
Chart Parsing

- Idea is to build up a table (known as a chart) of all calls to parse that have been made.

- Only one entry in chart for each distinct triple of arguments \((A: \alpha \bullet \beta, s, k)\).

- We’ll organize table in columns numbered by the \(k\) parameter, so that column \(k\) represents all calls that are looking at \(c_{k+1}\) in the input.

- Each column contains entries with the other two parameters: \([A: \alpha \bullet \beta, s]\), which are called items.

- The columns, therefore, are item sets.
Example

**Grammar**

\[
\begin{align*}
p &: e \, \llcorner \, \lnot, 0 \\
e &: s \, I \mid e \, \lnot \, \lnot \, e \\
s &: \lnot \, I \mid
\end{align*}
\]

**Input String**

\[-I + I \, \lnot\]

---

### Chart

Headings are values of \( k \) and \( c_{k+1} \) (raised symbols).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>I</td>
<td>2</td>
<td>+</td>
<td>3</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td>p:</td>
<td>•e , \lnot , \lnot , , 0</td>
<td>e.</td>
<td>s: \lnot , \lnot , \lnot , , 0</td>
<td>g.</td>
</tr>
<tr>
<td>b.</td>
<td>e:</td>
<td>•e \lnot , \lnot , e, 0</td>
<td></td>
<td>f.</td>
<td>e: s, I\lnot , , 0</td>
</tr>
<tr>
<td>c.</td>
<td>e:</td>
<td>•s , I, 0</td>
<td></td>
<td>h.</td>
<td>e: e \lnot , \lnot , \lnot , e, 0</td>
</tr>
<tr>
<td>d.</td>
<td>s:</td>
<td>•\lnot , \lnot , , 0</td>
<td></td>
<td></td>
<td>j.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>l.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n.</td>
<td>e:</td>
<td>e \lnot , \lnot , \lnot , e\lnot , , 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o.</td>
<td>p:</td>
<td>e\lnot , \lnot , \lnot , , 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Last modified: Wed Feb 16 10:38:52 2011
Example, completed

- Last slide showed only those items that survive and get used. Algorithm actually computes dead ends as well (unlettered, in red).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>I</th>
<th>2</th>
<th>+</th>
<th>3</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. p: ● e '⊣', 0</td>
<td>e. s: '⊣', 0, 0</td>
<td>g. e: s I●, 0</td>
<td>i. e: e '⊕' ● e, 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. e: ● e '⊕' e, 0</td>
<td>f. e: s ● I, 0</td>
<td>h. e: e ● '⊕' e, 0</td>
<td>j. e: ● s I, 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c. e: ● s I, 0</td>
<td>p: e ● '⊣', 0</td>
<td>k. s: ●, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d. s: ● '⊣', 0</td>
<td></td>
<td>l. e: s ● I, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s: ●, 0</td>
<td></td>
<td>s: ● '⊣', 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: s ● I, 0</td>
<td></td>
<td>e: ● e '⊕' e, 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>m. e: s I●, 3</td>
<td>p. p: e '⊣', ●, 0</td>
</tr>
<tr>
<td>n. e: e '⊕' e●, 0</td>
<td></td>
</tr>
<tr>
<td>o. p: e● '⊣', 0</td>
<td></td>
</tr>
<tr>
<td>e: e ● '⊕' e, 3</td>
<td></td>
</tr>
</tbody>
</table>
Adding Semantic Actions

• Pretty much like recursive descent. The call $\text{parse}(A: \alpha \cdot \beta, s, k)$ can return, in addition to $j$, the semantic value of the $A$ that matches characters $c_{s+1} \cdots c_j$.

• This value is actually computed during calls of the form $\text{parse}(A: \alpha' \cdot, s, k)$ (i.e., where the $\beta$ part is empty).

• Assume that we have attached these values to the nonterminals in $\alpha$, so that they are available when computing the value for $A$. 
Ambiguity

- Ambiguity only important here when computing semantic actions.
- Rather than being satisfied with a single path through the chart, we look at all paths.
- And we attach the set of possible results of \( \text{parse}(Y: \bullet \kappa, s, k) \) to the nonterminal \( Y \) in the algorithm.