Lecture 9: General and Bottom-Up Parsing
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 \( \alpha = \alpha_1 \alpha_n \ldots \alpha_n \) and each \( \alpha_i \) is a single terminal or nonterminal.

For example,

• \( A : \alpha \) might describe the production \( e : e ' + ' t \),
• \( B \Rightarrow \alpha A \gamma \Rightarrow \alpha \beta \gamma \) might describe the derivation steps \( e \Rightarrow e ' + ' t \Rightarrow e ' + ' ID \) (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

\[
A \xrightarrow{*} \underbrace{c_1 c_2 \ldots c_k C_{k+1} C_{k+2} \ldots C_n}_{A \xrightarrow{*}}
\]
Abstract body of parse(A,S)

- Can formulate top-down parsing analogously to NFAs.

```python
parse (A, S):
    """Assuming A is a nonterminal and S = c_1c_2...c_n is a string, return
    integer k such that A can derive the prefix string c_1...c_k of S."""
    Choose production ‘A: \(\alpha_1\alpha_2...\alpha_m\)’ for A (nondeterministically)
    k = 0
    for x in \(\alpha_1, \alpha_2, ..., \alpha_m\):
        if x is a terminal:
            if x == c_{k+1}:
                k += 1
            else:
                GIVE UP
        else:
            k += parse (x, c_{k+1}...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 `parse` returns a value if some set of choices for productions (the blue step) would return a value (just like NFA).

- Then if `parse(p, S)` returns a value, S must be in the language.
Example

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

\[
p : e \,'-'
\]
\[
e : t
\]
\[
\mid e \,'/\,t
\]
\[
\mid e \,'*\,t
\]
\[
t : \text{ID}
\]
Example

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

$$
\begin{align*}
& p : e '→' \\
& e : t \\
& \quad | e '/' t \\
& \quad | e '*' t \\
\end{align*}
$$

$t : \text{ID}$

$k_i$ means “the variable $k$ in the call to parse that is nested $i$ deep.” Outermost $k$ is $k_1$.

A failing path through the program:

```
p = p e '→'
```

Parse $p$:

```
p = p e '→'
```

```
Choose p : e '→':
```

Parse $e$:

```
e = t
```

```
Choose e : t:
```

Parse $t$:

```
t = \text{ID}
```

```
choose t : ID:
```

```
check S[1] == \text{ID}; OK, so k_3 += 1;
```

```
return 1 (= k_3; added to k_2)
```

```
return 1 (and add to k_1)
```

```
```

```
Example

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

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

$k_i$ means “the variable $k$ in the call to parse that is nested $i$ deep.” Outermost $k$ is $k_1$. Likewise for $S$.

A successful path through the program:

\[
\begin{align*}
  \text{parse}(p, S) & : \\
  \text{parse}(e, S) & : \\
  \text{parse}(e, S) & : \\
  \text{parse}(e, S) & :
\end{align*}
\]

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

  parse (A: $\alpha \bullet \beta$, s, k):

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

• Or diagrammatically, parse returns an integer j such that:

  $C_1 \cdots C_s \overbrace{C_{s+1} \cdots C_k}^{\alpha \rightarrow} \overbrace{C_{k+1} \cdots C_j}^{\beta \rightarrow} C_{j+1} \cdots C_n$
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 cs+1⋯ck.
Returns integer j such that β can produce ck+1⋯cj."""

if β is empty:
    return k

Assume β has the form xδ

if x is a terminal:
    if x == ck+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

\[
\begin{align*}
  p & : e \ '→' \\
  e & : s \ I \ | \ e \ '⁺' \ e \\
  s & : \ '‐' \ | \\
\end{align*}
\]

Input String

\[- \ I \ + \ I \ \ '→'
\]

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>1</td>
<td>I</td>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>a. p:  \bullet e \ '→' \ I, 0</td>
<td>e. s: \ '‐' \bullet, 0</td>
<td>g. e: s \bullet I, 0</td>
<td>i. e: e \ '⁺' \bullet e, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. e: \bullet e \ '⁺' \ e, 0</td>
<td>f. e: s\bullet I, 0</td>
<td>h. e: e \bullet \ '⁺' \ e, 0</td>
<td>j. e: \bullet s I, 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c. e: \bullet s I, 0</td>
<td></td>
<td></td>
<td>k. s: \bullet, 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d. s: \ '‐', 0</td>
<td></td>
<td></td>
<td></td>
<td>l. e: s \bullet I, 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>m. e: s \bullet I, 3</td>
<td></td>
<td>p. p: e \ '→' \bullet, 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n. e: e \ '⁺' \ e\bullet, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o. p: e\bullet \ '→', 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last modified: Sun Feb 8 18:07:31 2015

CS164: Lecture #9 10
**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></th>
<th>0</th>
<th>-</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:</td>
<td>⬤ e ’−’ , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. e:</td>
<td>⬤ e ’+’ e , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c. e:</td>
<td>⬤ s I , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d. s:</td>
<td>⬤ ’−’ , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s:</td>
<td>⬤ , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e:</td>
<td>s ⬤ I , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: e ’−’ , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: e ’+’ e , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: e ’+’ e , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: e ’+’ e , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: e ’+’ e , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: e ’+’ e , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e: e ’+’ e , 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Last modified: Sun Feb 8 18:07:31 2015
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 \( \text{parse}(A: \alpha', s, k) \) (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.