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 ($A, B, \ldots$).
- 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 ($\alpha, \beta, \ldots$).
- 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,
- $A: \alpha$ might describe the production $e: e '+' t$,
- $B \Rightarrow \alpha A \gamma \Rightarrow \alpha \beta \gamma$ might describe the derivation steps $e \Rightarrow e '+' t \Rightarrow e '+' ID$ ($\alpha$ is $e '+'$; $A$ is $t$; $B$ is $e$; and $\gamma$ 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

  $$
  \frac{c_1 c_2 \ldots c_k c_{k+2} \ldots c_n}{A \Rightarrow}
  $$

  

Abstract body of $\text{parse}(A, S)$

- Can formulate top-down parsing analogously to NFAs.
  
  ```python
  parse (A, S):
  """Assuming A is a nonterminal and S = c_1 c_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 += 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.
Consider parsing $S = \text{"ID*ID-"}$ with a grammar from last time:

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

A successful path through the program:

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

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

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.***
  
  if \beta is empty:
  return k

  Assume \beta has the form x\delta

  if x is a terminal:
  if x == c_{k+1}:
    return parse(A: x \bullet \delta, s, k+1)
  else:
    GIVE UP
  
  else:
    Choose production 'x: \kappa' for x (nondeterministically)
  
  j = parse(x: \bullet \kappa, k, k)
  
  return parse (A: \alpha \bullet \delta, 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 \((A: \alpha \cdot \beta, s, k)\).
- 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: \([A: \alpha \cdot \beta, s]\), which are called **items**.
- The columns, therefore, are **item sets**.

Example

**Grammar**

\[
p : e ' {-}' \\
e : s I \mid e ' +' e \\
s : ' {-}' \\
\]

**Input String**

- \(I + I \mid - I\)

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

<table>
<thead>
<tr>
<th>(k)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e)</td>
<td>p : e ' {-}'</td>
<td>0</td>
<td>e : s I</td>
<td>0</td>
<td>e : e ' +' e</td>
<td>0</td>
</tr>
<tr>
<td>(e)</td>
<td>b : e ' +' e</td>
<td>0</td>
<td>f : s I</td>
<td>0</td>
<td>g : e ' +' e</td>
<td>0</td>
</tr>
<tr>
<td>(c)</td>
<td>c : s I</td>
<td>0</td>
<td>s : e ' {-}'</td>
<td>0</td>
<td>j : e ' +' e</td>
<td>0</td>
</tr>
<tr>
<td>(d)</td>
<td>s : e ' {-}'</td>
<td>0</td>
<td>0</td>
<td>i : e ' +' e</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(s)</td>
<td>m : s I</td>
<td>0</td>
<td>n : e ' +' e</td>
<td>0</td>
<td>o : p : e ' {-}'</td>
<td>0</td>
</tr>
<tr>
<td>(e)</td>
<td>p : e ' {-}'</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Adding Semantic Actions

- Pretty much like recursive descent. The call \(parse(A: \alpha \cdot \beta, s, k)\) 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 \(parse(A: \alpha' \cdot, s, k)\) (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 $\text{parse}(Y: \bullet k, s, k)$ to the nonterminal $Y$ in the algorithm.