Topics in Computational Linguistics
— Parsing and Generation —

Stephan Oepen
Dan Flickinger

Center for the Study of Language and Information
Stanford University — Spring Quarter 2003

{oe|dan}@csli.stanford.edu

http://lingo.stanford.edu/courses/03/pg/
Review: Top-Down vs. Bottom-Up Parsing

Top-Down (Goal-Oriented)

- Left recursion (e.g. the ‘VP \rightarrow VP PP’ rule) causes infinite recursion;
- grammar conversion techniques (eliminating left recursion) exist, but will often be undesirable for natural language processing applications;

\[\rightarrow\] assume bottom-up as basic search strategy for remainder of the quarter.

Bottom-Up (Data-Oriented)

- unary (left-recursive) rules (e.g. ‘NP \rightarrow NP’) would still be problematic;
- lack of parsing goal: compute all possible derivations for, say, the input *adores snow*; however, it is ultimately rejected since it is not sentential;
- availability of partial analyses desirable for, at least, some applications.
Quantifying the Complexity of the Parsing Task

![Graph showing recursive function calls vs. number of prepositional phrases (n)]

<table>
<thead>
<tr>
<th>n</th>
<th>trees</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>46</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>170</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>593</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>2,093</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>7,539</td>
</tr>
<tr>
<td>5</td>
<td>132</td>
<td>27,627</td>
</tr>
<tr>
<td>6</td>
<td>429</td>
<td>102,570</td>
</tr>
<tr>
<td>7</td>
<td>1430</td>
<td>384,566</td>
</tr>
<tr>
<td>8</td>
<td>4862</td>
<td>1,452,776</td>
</tr>
</tbody>
</table>

*Kim adores snow (in Oslo)^n*
A Closer Look at the Calling Sequence

0: (bu-parse (kim adores snow in oslo))
  1: (bu-instantiate np (kim) nil (adores snow in oslo))
  2: (bu-instantiate S ([NP kim]) (VP) (adores snow in oslo))
  3: (bu-parse (adores snow in oslo))
  4: (bu-instantiate V (adores) nil (snow in oslo))
  5: (bu-instantiate VP ([V adores]) (NP) (snow in oslo))
      6: (bu-parse (snow in oslo))
          ...
  5: (bu-instantiate VP ([V adores]) nil (snow in oslo))
  6: (bu-instantiate VP ([VP [V adores]])) (PP) (snow in oslo))
      7: (bu-parse (snow in oslo))
          ...

2: (bu-instantiate NP ([NP kim]) (PP) (adores snow in oslo))
  3: (bu-parse (adores snow in oslo))
  4: (bu-instantiate V (adores) nil (snow in oslo))
  5: (bu-instantiate VP ([V adores]) (NP) (snow in oslo))
      6: (bu-parse (snow in oslo))
          ...
  5: (bu-instantiate VP ([V adores]) nil (snow in oslo))
  6: (bu-instantiate VP ([VP [V adores]]) (PP) (snow in oslo))
      7: (bu-parse (snow in oslo))
          ...

Parsing and Generation (35)
Memoization: Remember Earlier Results

Dynamic Programming

- The function call `(bu-parse (in oslo))` executes a total of ten times;
- memoization—record `bu-parse()` results for each starting string index;
  → use one-dimensional array, indexed by `from` value, for efficient access.

```lisp
(defun bu-parse (input from)
  (when (zerop from) (setf *chart* (make-array (length input))))
  (when input
    (or (aref *chart* from)
      (setf (aref *chart* from)
        (loop
          for rule in (rules-starting-in (first input))
          append ... ))))))
```

Parsing and Generation (36)
Our Weekly Dose of Lisp: Vectors and Arrays

- Multidimensional ‘grids’ of data can be represented as *vectors* or *arrays*;

- \((\text{make-array} \ (\text{rank}_1 \ldots \text{rank}_n))\) creates an array with \(n\) dimensions;

\[
\begin{align*}
\text{? (setf *foo* (make-array ’(2 5) :initial-element 0))} \\
\rightarrow #((0 0 0 0 0) (0 0 0 0 0))
\end{align*}
\]

\[
\begin{align*}
\text{? (setf (aref *foo* 1 2) 42)} \rightarrow 42
\end{align*}
\]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 42 & 0 & 0 \\
\end{array}
\]

- all dimensions count from zero; \(\text{aref}()\) accesses one individual cell;

- one-dimensional arrays are called *vectors* (abstractly similar to lists).
A Rerun: Local Variables

- Sometimes intermediate results need to be accessed more than once;
- `let()` and `let*()` create temporary value bindings for symbols, e.g;

  ```lisp
  ? (defparameter foo 42) → foo
  ? (let ((bar (+ foo 1))) bar) → 43
  ? bar → error
  ```

- bindings valid only in the body of `let()` (other bindings are shadowed);
- `let*()` binds sequentially, i.e. `variable_i` will be accessible for `variable_{i+1}`.
The Benefits of Dynamic Programming

Recursive Function Calls

- original algorithm
- memoized variant

Number of Prepositional Phrases ($n$)

Parsing and Generation (39)
Using the Chart to Bound Ambiguity

- For many substrings, multiple ways of deriving the same category;
- NPs: \(1 \mid 2 \mid 3 \mid 6 \mid 7 \mid 9\); PPs: \(4 \mid 5 \mid 8\); \(9 \equiv 1 + 8 \mid 6 + 5\);
- *parse forest* — a single item represents multiple trees [Billot & Lang, 89].
The CKY (Cocke, Kasami, & Younger) Algorithm

for (0 ≤ i < |input|) do
  $chart_{i,i+1} \leftarrow \{\alpha \mid \alpha \rightarrow input_i\}$;
for (0 ≤ l < |input|) do
  for (0 ≤ i < |input| − l) do
    for (1 ≤ j ≤ l) do
      if ($\alpha \rightarrow \beta_1 \beta_2 \in P \land \beta_1 \in chart_{i,i+j} \land \beta_2 \in chart_{i+j,i+l+1}$) then
        $chart_{i,i+l+1} \leftarrow chart_{i,i+l+1} \cup \{\alpha\}$;

\[
\begin{array}{cccccc}
0 & NP & S & S \\
1 & V & VP & VP \\
2 & NP & NP \\
3 & P & PP \\
4 & NP \\
\end{array}
\]

$[0,2] \leftarrow [0,1] + [1,2]$
$\ldots$
$[0,5] \leftarrow [0,1] + [1,5]$
$[0,5] \leftarrow [0,2] + [2,5]$
$[0,5] \leftarrow [0,3] + [3,5]$
$[0,5] \leftarrow [0,4] + [4,5]$

Parsing and Generation (41)
Generalized Chart Parsing

- The chart is a two-dimensional matrix of edges (aka chart items);
- an edge is a (possibly partial) rule instantiation over a substring of input;
- the chart indexes edges by start and end string position (aka vertices);
- dot in rule RHS indicates degree of completion: $\alpha \rightarrow \beta_1...\beta_i\cdot\beta_{i+1}...\beta_n$
- active edges (aka incomplete items) — partial RHS: $[1, 2, VP \rightarrow V \bullet NP]$;
- passive edges (aka complete items) — full RHS: $[1, 3, VP \rightarrow V NP\bullet]$;

The Fundamental Rule

$$[x, y, \alpha \rightarrow \beta_1...\beta_i\cdot\beta_{i+1}...\beta_n] + [y, z, \beta_i \rightarrow \gamma^+\bullet]$$

$$\leftrightarrow [x, z, \alpha \rightarrow \beta_1...\beta_i \cdot \beta_{i+1}...\beta_n]$$
An Example of a (Near-)Complete Chart

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NP → NP • PP</td>
<td></td>
<td></td>
<td></td>
<td>S → NP VP •</td>
</tr>
<tr>
<td></td>
<td>S → NP • VP</td>
<td></td>
<td></td>
<td></td>
<td>S → NP VP •</td>
</tr>
<tr>
<td></td>
<td>NP → kim •</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>VP → V • NP</td>
<td>VP → VP • PP</td>
<td>VP → VP • PP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>V → adores •</td>
<td>VP → V NP •</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>NP → NP • PP</td>
<td>NP → NP • PP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NP → snow •</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>PP → P • NP</td>
<td>PP → P NP •</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P → in •</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NP → NP • PP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NP → oslo •</td>
</tr>
</tbody>
</table>

0 Kim 1 adores 2 snow 3 in 4 Oslo 5
Suggested Background Activities

- Retrieve the model solution for the second exercise from the course site;
- compare our solution to your submission; how is ours better (or not)?
- draw the chart from slide # 43 in the more accessible graph format (# 40);
- write nested loop(s) to reflect the CKY control structure (slide # 41);
- have a look at Chapters Nine and Ten of [Jurafsky & Martin, 2000];
- re-read Chapters Thirteen, and Fifteen of [Winston & Horn, 1989].