Natuurlijke Taalverwerking
Natural Language Processing

Gosse Bouma and Begoña Villada

3e trimester 2002/2003
Overview

1. Parsing
   - Left-recursion,
   - Top-down vs. Bottom-up strategies,

2. Shift-reduce parsing
   - Implementation in Prolog,
   - Ambiguity,
   - Epsilon-rules.
Parsing

- Prolog provides top-down, depth-first, parsing strategy as default,
- Many alternative strategies exist,
- Often more robust and efficient.
Top-down Parsing

- DCG uses Prolog top-down search strategy,
- Therefore, left-recursion leads to problems,

\[
\text{ancestor}(X,Y) : - \\
\quad \text{parent}(X,Y). \\
\text{ancestor}(X,Y) : - \\
\quad \text{ancestor}(X,Z), \text{parent}(Z,Y).
\]
Left-recursion in Grammar

- een kind, een kind in het park
- n --> n, pp.
- Kim slaapt, Kim slaapt tot 10 uur
- vp --> vp, pp
- Kim slaapt en Sandy werkt
- s --> s, [en], s.
Left-recursion in Grammar

- Peter’s (broer’s) huis
- np --> det, n
- det --> np, [s].
- een (erg) aardig kind
- a --> int, a.
- int --> [erg];[heel];[vet]; [].

•
Removing Left-recursion

\[ n \rightarrow n, \ pp \]
\[ pp \rightarrow p, \ n \]
\[ n \rightarrow n_{\text{lex}}, \ pp_{\star} \]
\[ n_{\text{lex}} \rightarrow [\text{kind}] \]
\[ pp_{\star} \rightarrow pp, \ pp_{\star} \]
\[ pp_{\star} \rightarrow [] \]
\[ pp \rightarrow p, \ np \]
Removing Left-recursion

• Changing the grammar also changes the structure assigned to input-strings,

• Although this can be fixed as well.
Removing Left-recursion

- Removing left-recursion can lead to explosion of the number of rules in the grammar. (Moore, Proceedings NAACL, 2000)

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Number of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toy</td>
<td>88</td>
</tr>
<tr>
<td>Paull “Best”</td>
<td>156</td>
</tr>
<tr>
<td>Paull “Lexicographic”</td>
<td>970</td>
</tr>
<tr>
<td>Paull “Worst”</td>
<td>5696</td>
</tr>
</tbody>
</table>
Alternative Parsing Methods

- Prolog searches top-down, depth-first,

- Alternatives:
  - **Bottom-up Parsing**: work from the input towards the goal (s).
  - **Breadth-first (parallel)**: Explore all ways to expand a rule in parallel.
Alternative Parse Strategy

- Requires separation of rules (data) and parser (algorithm)

- Grammar rules as data:

  \% rule(Mother\_Category,List\_Daughters)
  rule(s,[np,vp]).

  \% lex(Category\_Word).
  lex(np,kim).
Top-down Parsing in Prolog

top_down(Cat,P0,P1) :-
    rule(Cat,Daughters),
    find_ds(Daughters,P0,P1).

break

top_down(Cat,[Word|Ws],Ws) :-
    lex(Cat,Word).

find_ds([D1|Ds],P0,P2) :-
    top_down(D1,P0,P1),
    find_ds(Ds,P1,P2).

find_ds([],P0,P0).
# Top-down vs Bottom-up

<table>
<thead>
<tr>
<th>Top-down Parsing</th>
<th>Bottom-up Parsing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td>the dog barks</td>
</tr>
<tr>
<td>NP VP</td>
<td>DET dog barks</td>
</tr>
<tr>
<td>DET N VP</td>
<td>DET N barks</td>
</tr>
<tr>
<td>the N VP</td>
<td>NP barks</td>
</tr>
<tr>
<td>the dog VP</td>
<td>NP V</td>
</tr>
<tr>
<td>the dog V</td>
<td>NP VP</td>
</tr>
<tr>
<td>the dog barks</td>
<td>S</td>
</tr>
</tbody>
</table>
Shift-reduce parsing

- **Bottom-up parsing!**
- Start with the input, and search for lexical categories,
- Try to combine categories into phrases
- Try to combine phrases into larger phrases or a sentence.
- Bottom-up parsers do not loop on left-recursive rules.
Shift-reduce Algorithm

- **Stack**: for storing intermediate result,
- **Shift**: Remove the leftmost element of the input and add its category to the Stack,
- **Reduce**: Replace $C_1..C_n$ on the Stack by $C_0$ given a rule $C_0 \rightarrow C_1..C_n$. 
## Shift-reduce Algorithm

<table>
<thead>
<tr>
<th></th>
<th>String</th>
<th>Stack</th>
<th>Action</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>the dog barks</td>
<td>[]</td>
<td>sh</td>
<td>lex(det,the)</td>
</tr>
<tr>
<td>2</td>
<td>dog barks</td>
<td>[det]</td>
<td>sh</td>
<td>lex(n,dog)</td>
</tr>
<tr>
<td>3</td>
<td>barks</td>
<td>[det,n]</td>
<td>red</td>
<td>rule(np,[det,n])</td>
</tr>
<tr>
<td>4</td>
<td>barks</td>
<td>[np]</td>
<td>sh</td>
<td>lex(v,barks)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>[np,v]</td>
<td>red</td>
<td>rule(vp,[v])</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>[np,vp]</td>
<td>red</td>
<td>rule(s,[np,vp])</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>[s]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Shift-reduce in Prolog

sr(Input,Stack) :-
    reduce(Stack,NewStack),
    sr(Input,NewStack).

sr(Input,Stack) :-
    shift(Input,Rest,Cat),
    sr(Rest,[Cat|Stack]).

shift([Wrd|Input],Input,Cat) :-
    lex(Cat,Wrd).
Shift-reduce in Prolog

reduce(Stack, [M|NewStack]) :-
    reduce_rule(M,Ds),
    append(Ds, NewStack, Stack).

reduce_rule(s, [vp, np]).

• Note the order of Ds in reduce_rule is reversed!
Optimized Reduce

reduce([vp,np|Stack],[s|Stack]).
reduce([n,det|Stack],[np|Stack]).

• No need for append or search,

• Rules can be automatically converted in reduce actions.
Ambiguity

- Kim bought a house with a garage.
- Kim bought a house in January.
**Shift-reduce conflict**

<table>
<thead>
<tr>
<th>String</th>
<th>Stack</th>
<th>action</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. bought a house with a garage</td>
<td>[]</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>m. with a garage</td>
<td>[v,np]</td>
<td>red</td>
<td>vp → v np</td>
</tr>
<tr>
<td>n. with a garage</td>
<td>[vp]</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>o.</td>
<td>[vp,pp]</td>
<td>red</td>
<td>vp → vp pp</td>
</tr>
<tr>
<td>m. with a garage</td>
<td>[v,np]</td>
<td>shift</td>
<td>lex(with,p)</td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p.</td>
<td>[v,np,pp]</td>
<td>red</td>
<td>np → np pp</td>
</tr>
<tr>
<td>q.</td>
<td>[v,np]</td>
<td>..</td>
<td></td>
</tr>
</tbody>
</table>
Main and Subordinate Clauses

- Piet slaapt
- Jan denkt dat Piet slaapt
- Piet leest een boek
- Jan denkt dat Piet een boek leest

\[
\begin{align*}
  s & \rightarrow \text{np vp} & \text{bijzin} & \rightarrow \text{np vpb} \\
  \text{vp} & \rightarrow \text{v np} & \text{vpb} & \rightarrow \text{np v} \\
  \text{vp} & \rightarrow \text{v} & \text{vpb} & \rightarrow \text{v} \\
  \text{vp} & \rightarrow \text{v [dat] bijzin}
\end{align*}
\]
## Reduce-reduce conflict

<table>
<thead>
<tr>
<th>String</th>
<th>Stack</th>
<th>Action</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Jan denkt dat Piet slaapt</td>
<td>[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m.</td>
<td>[..,dat,np,vp]</td>
<td>red</td>
<td>s $\rightarrow$ np vp</td>
</tr>
<tr>
<td>n.</td>
<td>[..dat,s]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m.</td>
<td>[..,dat,np,vp]</td>
<td>red</td>
<td>bz $\rightarrow$ np vpb</td>
</tr>
<tr>
<td>n</td>
<td>[..dat,bz]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Advantages of shift-reduce Parsing

- Left-recursion is no problem.
- Size of the stack bounded by the number of words in the input.
- Reduce-actions terminate as long as the grammar contains no cycles.

\[
\begin{align*}
\text{np} & \rightarrow n \\
n & \rightarrow np
\end{align*}
\]
Disadvantage of Shift-reduce

\[ \text{det} \rightarrow \epsilon \]

- Epsilon-rules require that a category is added to the stack, which does not correspond to a word in the input.
- Size of the stack no longer bounded by the input.
- Epsilon-rules cause non-termination.
SR and Epsilon’s

<table>
<thead>
<tr>
<th>String</th>
<th>Stack</th>
<th>action</th>
<th>rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>dogs bark</td>
<td>[]</td>
<td>sh</td>
<td></td>
</tr>
<tr>
<td>dogs bark</td>
<td>[det]</td>
<td>sh</td>
<td></td>
</tr>
<tr>
<td>dogs bark</td>
<td>[det, det]</td>
<td>sh</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Removing Epsilon’s

• The effect of epsilon-rules can be achieved indirectly as well:

\[ np \rightarrow \text{det } n \quad \text{det} \rightarrow \epsilon \]

\[ np \rightarrow n \]
Removing Epsilon’s

- For all rules $C \rightarrow \epsilon$ and
- all rules $M \rightarrow C_1...C_i, C, C_j...C_n$,
- Add $M \rightarrow C_1...C_i, C_j...C_n$. 
Efficiency

- DCG and shift-reduce parsers are normally depth-first parsers:
  - Find different solutions by backtracking,
- Depth-first parsing can be very inefficient,
- Breadth-first parsing is usually much more efficient,
- Chart-parsers are Breadth-first parsers.