Lecture 17: Hybrid Algorithms: Left Corner Parsing

CSA3202 Human Language Technology

Mike Rosner, Dept ICS
December 2011

Contents

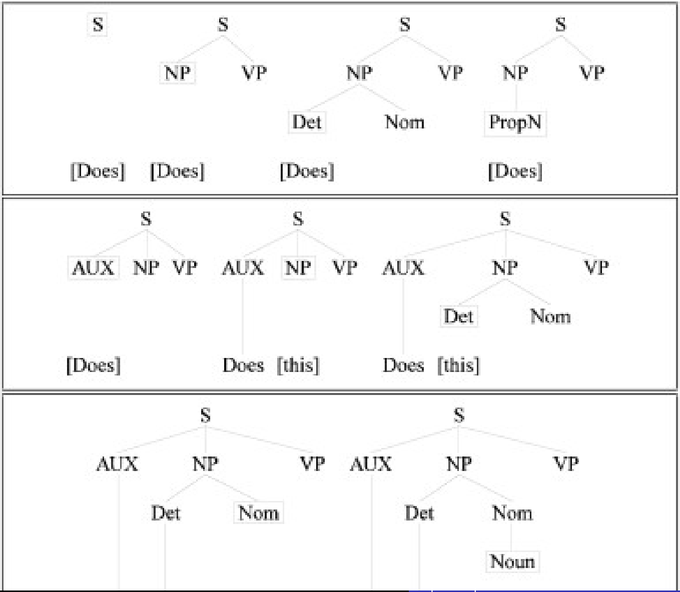
1 Motivation for Hybrid Algorithms

2 Left Corner Parsing

Motivation

- Disadvantages of pure Top Down
  - Left Recursion
  - Repeated Work
- Disadvantages of pure Bottom Up
  - Build useless structures which will never be derivable from S
  - Rules which introduce empty categories
- Chart parser (both Earley and CKY) both address repeated work issue by storing intermediate results in a table.
- Left Corner parser - hybrid strategy

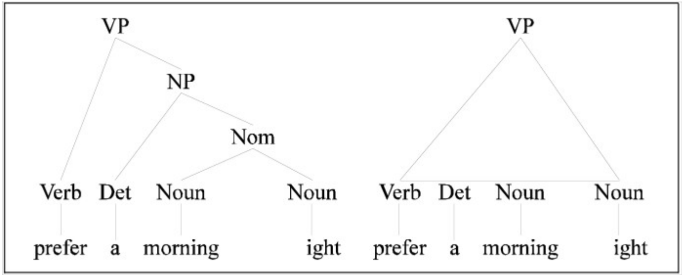
Top Down Recursive Descent



Avoiding Useless Top Down Predictions

- We know the current input word must serve as the first word in the derivation of the unexpanded node the parser is currently processing.
- Therefore the parser should not consider grammar rule for which the current word cannot serve as the "left corner"
- The left corner we are interested in is the first preterminal node along the left edge of a derivation.

Example of a Left Corner



What is a Left Corner?

- We define the relation $\angle$ between nonterminals such that $B \angle A$ if and only if there is a rule $A \rightarrow B\alpha$, where α denotes some sequence of grammar symbols.
- The transitive and reflexive closure of $\angle$ is denoted by $\angle^*$, which is called the left-corner relation.
- Informally, we have that $B \angle^* A$ if and only if it is possible to have a spine in some parse tree in which B occurs below A (or B = A).
- Possible left corners of all non-terminal categories can be determined in advance and placed in a table.

Left Corner Table

```
s    --> np, vp.
s   --> aux, np, vp.
s   --> vp.
np   --> det, nom.
np   --> pn.
nom   --> noun.
nom   --> noun, nom.
nom   --> nom, pp
pp   --> prep, np.
vp   --> v.
vp   --> v np.
```

Category	Left Corners	Category	Left Corners
s	np pn, det, aux, v	vp	v
np	det, pn, nom, noun	pp	prep
nom	nom, noun		

Left Corner: Key Idea

- Key Idea: accept a word, identify the constituent it marks the beginning of, and parse the rest of the constituent top down.
- Main Advantages:
  - Like a bottom-up parser, can handle left recursion without looping, since it starts each constituent by accepting a word from the input string.
  - Like a top-down parser, is always expecting a particular category for which only a few of the grammar rules are relevant. It is therefore more efficient than a plain shift-reduce algorithm.

Left Corner Parsing - Basic Idea

- Left-corner combines bottom-up and top-down strategies in the following sense.
- Given a rule: $k_0 \longrightarrow k_1 k_2 \ldots k_n$
- Normal bottom-up: all k1 to kn must be recognized before applying the rule
- Left-corner: it suffices that k_1 is recognized
- k2 to kn and the dominating nodes of k_1 are predicted top-down

Left Corner Algorithm

To parse a constituent of type C:

1. Accept a word W from input and determine K, its category.
2. Complete C:
    - If K=C, exit with success; otherwise
    - Find a constituent whose expansion begins with K. Call that CC. For instance, if K=d (determiner), CC could be Np, since we have rule(np,[d,n])
    - Recursively left-corner parse all the remaining elements of the expansion of CC (in this case, [n]).
    - Put CC in place of K, and return to step 2

Left Corner Recogniser in Prolog

```
parse(C,[W|Rest],P) :-
 word(K,W),
 complete(K,C,Rest,P).

parse_list([],P,P).
parse_list(([C|Cs],P1,P) :-
 parse(C,P1,P2),
 parse_list(Cs,P2,P).

complete(C,C,P,P).              % if C=W, do nothing
complete(K,C,P1,P) :-
 rule(CC,[K|Rest]),
 parse_list(Rest,P1,P2),
 complete(CC,C,P2,P).
```

Trace of Left Corner Parse

```
Call:  (  7) parse(np, [the, cat], []) ? creep
Call:  (  8) word(_L128, the) ? creep
Exit:  (  8) word(d, the) ? creep
Call:  (  8) complete(d, np, [cat], []) ? creep
Call:  (  9) rule(_L153, [d|_G306]) ? creep
Exit:  (  9) rule(np, [d, n]) ? creep
Call:  (  9) parse_list([n], [cat], _L155) ? creep
Call:  ( 10) parse(n, [cat], _L181) ? creep
Call:  ( 11) word(_L196, cat) ? creep
Exit:  ( 11) word(n, cat) ? creep
Call:  ( 11) complete(n, n, [], _L181) ? creep
Exit:  ( 11) complete(n, n, [], []) ? creep
Exit:  ( 10) parse(n, [cat], []) ? creep
Call:  ( 10) parse_list([], [], _L155) ? creep
Exit:  ( 10) parse_list([], [], []) ? creep
Exit:  (  9) parse_list([n], [cat], []) ? creep
Call:  (  9) complete(np, np, [], []) ? creep
Exit:  (  9) complete(np, np, [], []) ? creep
```

References