October 14, 2003

Chapter 10.1–10.2

CFGs, Parsing

What is Parsing?

- Assigning (syntactic) structure to input strings
- Here: Based on context-free grammars
- What other knowledge systems could a parser use?

Context-Free Grammar

- A CFG is a quadruple: $\langle C, \Sigma, P, S \rangle$:
  - C is the set of *categories* (aka *non-terminals*, e.g., { S, NP, VP, V };
  - Σ is the *vocabulary* (aka *terminals*), e.g., { Kim, snow, adores }
  - P is a set of *rewrite rules* of the form $\alpha \rightarrow \beta_1, \beta_2, \ldots, \beta_n$
  - $S \in C$ is the *start-symbol*
  - For each rule ‘$\alpha \rightarrow \beta_1, \beta_2, \ldots, \beta'_n \in P : \alpha \in C \text{and} \beta_i \in C \cup \Sigma; 1 \leq i \leq n$.

Context-Free Languages

- CFGs are more powerful than regular expressions/FSAs – that is, the former can generate languages that the latter can't.
- A case in point: $a^n B^n$
  - S $\rightarrow$ a S b
  - S $\rightarrow \epsilon$

Context-Free Languages

- There are languages CFGs can't generate (non-context-free languages), notably those that incorporate *cross-serial dependencies*, such as Swiss German.
- Perhaps more importantly, CFGs are cumbersome and inefficient for representing natural language syntax.
- Most (but not all) modern theories of syntax include a notion of phrase structure (CFG), and then extend it.

Toy CFG

- C: { S, VP, PP, NP, V, P }
- Σ: { Kim, snow, Oslo, adores, in, snores }
- P: (see next slide)
- S: S

Toy CFG

S → NP VP

VP → V NP

VP → V

VP → VP PP

PP → P NP

NP → NP PP

NP → Kim

NP → snow

NP → Oslo

V → adores

V → snores

P → in

Human (deliberate) parsing

- What tree does the toy CFG assign to this sentence:

  Kim adores snow in Oslo

- How did you determine that?

Four parameters of parsing algorithms

- Top-down v. bottom-up
- Breadth-first v. depth-first
- Best-first v. exhaustive
- Uni- v. bi-directional

A basic top-down parser (p.366)

```
function TOP-DOWN-PARSE(input,grammar) returns a parse tree
    agenda ← (Initial S tree, Beginning of input)
    css ← POP(agenda)
    loop
        if SUCCESSFUL-PARSE?(css) then
            return TREE(css)
        else
            if CAT(NODE-TO-EXPAND(css)) is a POS then
                if CAT(node-to-expand) ⊂ POS(CURRENT-INPUT(css)) then
                    PUSH(APPLY-LEXICAL-RULE(css),agenda)
                else
                    PUSH(APPLY-RULES(css,grammar),agenda)
            if agenda is empty then
                return reject
            else
                css ← NEXT(agenda)
    end
```

Problems with this algorithm

- Infinite loops with left-recursive grammars
- Ambiguity
- Inefficient reparsing of subtrees
- ... solve all three with dynamic programming: chart parsing.

A note on the homework

- Assignment 2 will probably be easier than Assignment 1
- That doesn't mean you should get started soon!
- The parsing section of Assignment 2 has you examining the final state of the chart in a chart parser after it has parsed various sentences.
- The LKB chart parser is not an implementation of the Earley chart parser. It only stores completed (or 'passive') edges.