General Context-Free Grammar Parsing
A phrase structure grammar

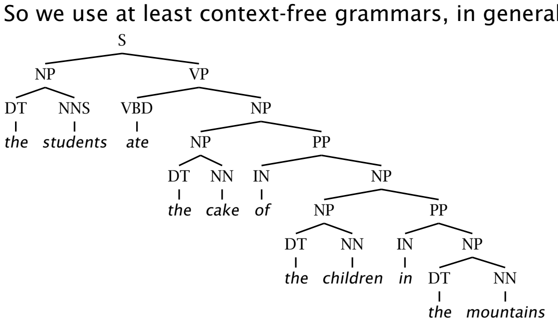
- Also known as a context-free grammar (CFG)
- S → NP VP
  NP → {DT NNS, DT NN, NP PP}
  VP → {VP PP, VBD, VBD NP}
  PP → IN NP
  DT → *the*
  NNS → {*children*, *students*, *mountains*}
  VBD → {*slept*, *ate*, *saw*}
  IN → {*in*, *of*}
  NN → *cake*

Application of grammar rewrite rules

- S
  → NP VP
  → DT NNS VBD
  → *The children slept*
- S
  → NP VP
  → DT NNS VBD NP
  → DT NNS VBD DT NN
  → *The children ate the cake*

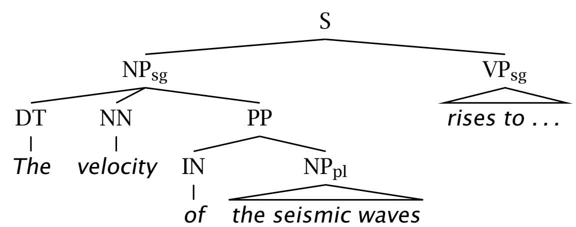
Phrase structure is recursive

So we use at least context-free grammars, in general



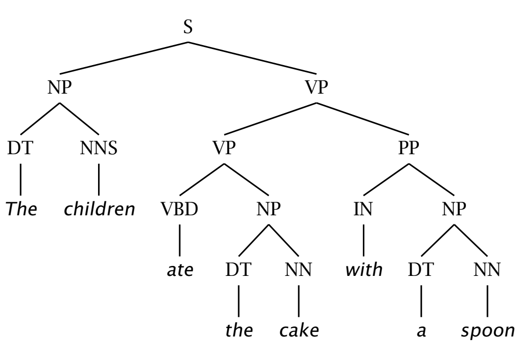
Why we need recursive phrase structure

- Kupiec (1992): Sometimes HMM tagger goes awry: *waves* → verb
- *The velocity of the seismic waves rises to . . .*

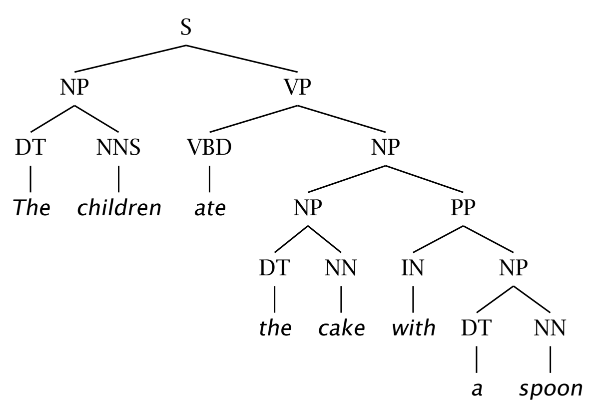


- Language model: There are similar problems.
  The captain of the ship yelled out.

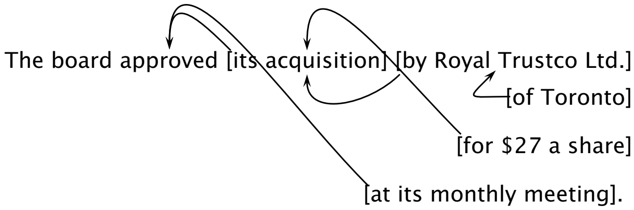
Natural language grammars are ambiguous:
Prepositional phrase verb attachment



PP Ambiguity: NP attachment



Attachment ambiguities in a real sentence



Ambiguity

- Programming language parsers resolve local ambiguities with lookahead
- Natural languages have global ambiguities:
  - *I saw that gasoline can explode*
- Size of embedded NP?

What is parsing?

- We want to run the grammar backwards to find the structures
- Parsing can be viewed as a search problem
- Parsing is a hidden data problem
- We search through the legal rewritings of the grammar
- We want to find *all* structures for a string of words (for the moment)
- We can do this bottom-up or top-down
  - This distinction is independent of depth-first/bread-first etc. – we can do either both ways
  - Doing this we build a *search tree* which is different from the *parse tree*

State space search

- States:
- Operators:
- Start state:
- Goal test:
- *Algorithm*

```
Put start state on stack
solutions = {}
loop
      if stack is empty, return solutions
      state = remove-front(stack)
      if goal(state) push(state, solutions)
      stack = push(expand(state, operators), nodes)
end
```


Another phrase structure grammar

S	→	NP VP	N	→	*cats*
VP	→	V NP	N	→	*claws*
VP	→	V NP PP	N	→	*people*
NP	→	NP PP	N	→	*scratch*
NP	→	N	V	→	*scratch*
NP	→	e	P	→	*with*
NP	→	N N	PP	→	P NP

cats scratch people with claws

S					
NP	VP				
NP	PP	VP			*3 choices*
NP	PP	PP	VP		
oops!					
N	VP				
cats	VP				
cats	V	NP			*2 choices*
cats	scratch	NP			
cats	scratch	N			*3 choices – showing 2nd*
cats	scratch	people			*oops!*
cats	scratch	NP	PP		
cats	scratch	N	PP		*3 choices – showing 2nd . . .*
cats	scratch	people	with	claws	

Phrase Structure (CF) Grammars

$$G = \langle T, N, S, R \rangle$$

- T is set of terminals
- N is set of nonterminals
  - For NLP, we usually distinguish out a set $P \subset N$ of *preterminals* which always rewrite as terminals
- S is start symbol (one of the nonterminals)
- R is rules/productions of the form $X \rightarrow \gamma$, where X is a nonterminal and γ is a sequence of terminals and nonterminals (may be empty)
- A grammar G generates a language L

Recognizers and parsers

- A *recognizer* is a program for which a given grammar and a given sentence returns **yes** if the sentence is accepted by the grammar (i.e., the sentence is in the language) and **no** otherwise
- A *parser* in addition to doing the work of a recognizer also returns the set of parse trees for the string

Soundness and completeness

- A parser is *sound* if every parse it returns is valid/correct
- A parser *terminates* if it is guaranteed to not go off into an infinite loop
- A parser is *complete* if for any given grammar and sentence it is sound, produces every valid parse for that sentence, and terminates
- (For many purposes, we settle for sound but incomplete parsers: e.g., probabilistic parsers that return a k-best list)

Top-down parsing

- Top-down parsing is goal directed
- A top-down parser starts with a list of constituents to be built. The top-down parser rewrites the goals in the goal list by matching one against the LHS of the grammar rules, and expanding it with the RHS, attempting to match the sentence to be derived.
- If a goal can be rewritten in several ways, then there is a choice of which rule to apply (search problem)
- Can use depth-first or breadth-first search, and goal ordering.

Bottom-up parsing

- Bottom-up parsing is data directed
- The initial goal list of a bottom-up parser is the string to be parsed. If a sequence in the goal list matches the RHS of a rule, then this sequence may be replaced by the LHS of the rule.
- Parsing is finished when the goal list contains just the start category.
- If the RHS of several rules match the goal list, then there is a choice of which rule to apply (search problem)
- Can use depth-first or breadth-first search, and goal ordering.
- The standard presentation is as *shift-reduce* parsing.

Problems with top-down parsing

- Left recursive rules
- A top-down parser will do badly if there are many different rules for the same LHS. Consider if there are 600 rules for S, 599 of which start with NP, but one of which starts with V, and the sentence starts with V.
- Useless work: expands things that are possible top-down but not there
- Top-down parsers do well if there is useful grammar-driven control: search is directed by the grammar
- Top-down is hopeless for rewriting parts of speech (preterminals) with words (terminals). In practice that is always done bottom-up as lexical lookup.
- Repeated work: anywhere there is common substructure

Problems with bottom-up parsing

- Unable to deal with empty categories: termination problem, unless rewriting empties as constituents is somehow restricted (but then it's generally incomplete)
- Useless work: locally possible, but globally impossible.
- Inefficient when there is great lexical ambiguity (grammar-driven control might help here)
- Conversely, it is data-directed: it attempts to parse the words that are there.
- Repeated work: anywhere there is common substructure
- Both TD (LL) and BU (LR) parsers can (and frequently do) do work exponential in the sentence length on NLP problems.

Principles for success: what one needs to do

- Left recursive structures must be found, not predicted
- Empty categories must be predicted, not found

An alternative way to fix things

- Grammar transformations can fix both left-recursion and epsilon productions
- Then you parse the same language but with different trees
- Linguists tend to hate you.
- But they shouldn't providing you can fix the trees post hoc.