CS 181: Natural Language Processing

Lecture 9: Context Free Grammars

Kim Bruce
Pomona College
Spring 2008

Disclaimer: Slide contents borrowed from many sources on web!

Homework & NLTK

- Review MLE, Laplace, and Good-Turing in smoothing.py

Motivation

- Chunks of sentences behave as units
- Want to recover from input.
- Reason: Chunks are basis of meaning
- Subtrees of parse trees will represent meaningful chunks for us.

Formal Def of CFG

- G = <T, N, S, R>, where
  - T is a set of *terminals* (lexicon)
  - N is a set of *non-terminals*. In linguistics, often also identify $P \subseteq N$, *preterminals*, which always rewrite as terminals.
  - $S \in N$ is *start state*.
  - R is set of rules of form $X \rightarrow \gamma$, where X is non-terminal and γ is sequence composed of terminals and non-terminals.
- $L(G) = \{w \in T^* \mid S \rightarrow^* w\}$

Uses of CFG

- Generate sentences of language
- Recognize sentences of language
  - Impose structure as well!
  - Sentences in language said to be grammatical

Example CFG

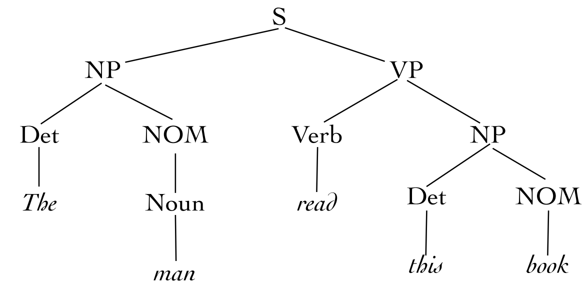
- T = {this, that, a, the, man, book, flight, meal, include, read, does}
- N = {S, NP, NOM, VP, Det, Noun, Verb, Aux}
- S - start
- R =

S → NP VP	VP → Verb
S → Aux NP VP	VP → Verb NP
S → VP	Det → that | this | a | the
NP → Det NOM	Noun → book | flight | meal | man
NOM → Noun	Verb → book | include | read
NOM → Noun NOM	Aux → does

Derivation

S
→ NP VP
→ Det NOM VP
→ The NOM VP
→ The Noun VP
→ The man VP
→ The man Verb NP
→ The man read NP
→ The man read Det NOM
→ The man read this NOM
→ The man read this Noun
→ The man read this book

Parse Tree



Many derivations give the same tree.
Abstracts away order!

CFG's & Recursion

- Non-trivial recursion represented nicely with cfg's:
  - NP → NP PP
  - PP → Prep NP
- [$_{S}$[$_{NP}$The student] [$_{VP}$ [$_{VB}$ took] [$_{NP}$ a class [$_{PP}$ in [$_{NP}$ Edmunds]]] [$_{PP}$ with [$_{NP}$ her friend]]]]   *bracket notation represents tree*
- Rule like VP → V NP allows to ignore internal complexity of NP.

Polymorphism

- S → S *and* S
  - John liked Mary and she liked him
- NP → NP *and* NP
  - John like Mary and Suzy
- VP → VP *and* VP
- ...
- Need X → X *and* X
  - Any restrictions?

Unwanted Complexity

- Agreement
  - He plays on the swings.
  - They play on the swings.
  - ... this flight
  - ... these flights
- Other languages require gender
- Subject vs. object (he vs. him)

Subcategorization

- Verbs with objects and indirect objects
  - John sneezed.
  - Mary found [$_{NP}$ a phone].
  - Jane gave [$_{NP}$ the teacher] [$_{NP}$ an apple].
  - I prefer [$_{TO\_VP}$ to do it myself]
  - I was told [$_{S}$ it is not allowed to litter].
- Subcategorization expresses constraints on a word on the number and type of arguments associated with it.

Allows Incorrect Sentences

- VP → V NP
  - Allows "John sneezed the book."
  - Distinguish between transitive & non-transitive
    - *but many more distinctions!*
  - Subcategorization frames
- Can complicate the grammar or add other mechanisms (non-cfg) to take care of these issues.
- Come back to it later!

Movement

- [$_S$ [$_{NP}$ My travel agent][$_{VP}$ booked [$_{NP}$ the flight]]]
- [$_S$ [$_{NP}$ Which flight] do you want me to have the travel agent [$_V$ book]]?
- *Separated object (the flight) from verb (book)*
- *How can we recover constituents?*

Equivalence of Grammars

- Two grammars are:
  - *strongly equivalent* if
    - they generate the same sentences
    - they assign the same structure to each sentence
  - *weakly equivalent* if
    - they generate the same sentences
    - they may not assign the same structure to each sentence
- Alas, weakly equivalent is undecidable!

Normal Forms

- Useful in performing algorithms on cfg's
  - Greibach
  - Chomsky (CNF)
    - Productions of form: A → B C or A → a or S → ε
- Theorem: Any cfg can be converted into a weakly equivalent grammar in CNF.

Conversion to CNF

- Make S non-recursive (add S' if necessary)
  - *Only necessary if ε is in language..*
- Eliminate all ε-moves except for S
- Eliminate unit productions (S → T).
- Reduce right hand sides to length 2.
- Convert all terminals on rt sides to non-terminals
- Remove any useless rules & symbols

MAKE S NON-RECURSIVE

- Convert:
  - S → ε
  - S → A B S
  - A → ε
  - A → x y z
  - B → w B
  - B → v
- Step 1:
  - S → ε
  - S → S'
  - S' → A B S'
  - S' → ε
  - A → ε
  - A → x y z
  - B → w B
  - B → v

Eliminate ε-moves

- Step 1:
  - S → ε
  - S → S'
  - S' → A B S'
  - S' → ε
  - A → ε
  - A → x y z
  - B → w B
  - B → v
- Step 2:
  - S → ε
  - S → S'
  - S' → A B S'
  - S' → A B
  - S' → B S'
  - S' → B
  - A → x y z
  - B → w B
  - B → v

Eliminate Unit Productions

- Step 2:
  - S → ε
  - S → S'
  - S' → A B S'
  - S' → A B
  - S' → B S'
  - S' → B
  - A → x y z
  - B → w B
  - B → v
- Step 3:
  - S → ε
  - S → A B S'
  - S → A B
  - S → B S'
  - S → w B
  - S → v
  - S' → A B S'
  - S' → A B
  - S' → B S'
  - S' → w B
  - S' → v
  - A → x y z
  - B → w B
  - B → v

Shrink RHS

- Step 3:
  - S → ε
  - S → A B S'
  - S → A B
  - S → B S'
  - S → w B
  - S → v
  - S' → A B S'
  - S' → A B
  - S' → B S'
  - S' → w B
  - S' → v
  - A → x y z
  - B → w B
  - B → v
- Step 4:
  - S → ε
  - S → A P
  - P → B S'
  - S → A B
  - S → B S'
  - S → w B
  - S → v
  - S' → A P
  - S' → A B
  - S' → B S'
  - S' → w B
  - S' → v
  - A → x Q
  - Q → y z
  - B → w B
  - B → v

Eliminate terminals

- Step 4:
  - S → ε
  - S → A P
  - P → B S'
  - S → A B
  - S → B S'
  - S → w B
  - S → v
  - S' → A P
  - S' → A B
  - S' → B S'
  - S' → w B
  - S' → v
  - A → x Q
  - Q → y z
  - B → w B
  - B → v
- Step 5:
  - S → ε
  - S → A P
  - P → B S'
  - S → A B
  - S → B S'
  - S → W B
  - S → v
  - S' → A P
  - S' → A B
  - S' → B S'
  - S' → W B
  - S' → v
  - A → X Q
  - Q → Y Z
  - B → W B
  - B → v
  - W → w
  - X → x
  - Y → y
  - Z → z

Parsing

- Assign parse trees to legal sentence of the language.
- Regular languages (expressions) can be recognized by FSA in linear time.
- Can't represent languages like a^n b^n.
- Cfg's can't represent a^n b^n c^n.
  - Requires context-sensitive.

Parsing CFG's

- Requires push-down automaton.
  - FSA w/ stack to hold partial results
- Complexity
  - space to parse - O(n) (proportional to recursion),
  - time $O(n^3)$

Parsing

- Recognizer just says yes or no
- Parser: Given term, find parse tree
  - Bottom-up
  - Top-down
- Want to find all parse trees! *Ambiguity!*
- Want to determine which is most likely
- Short-term memory in humans is limited

Empty Rules and Left Recursion

- Grammars can
  - have an ε-rule: A → ε
  - have left-recursive rules: A → AB
    - E.g. VP → VP PP
- Make parsing more difficult!

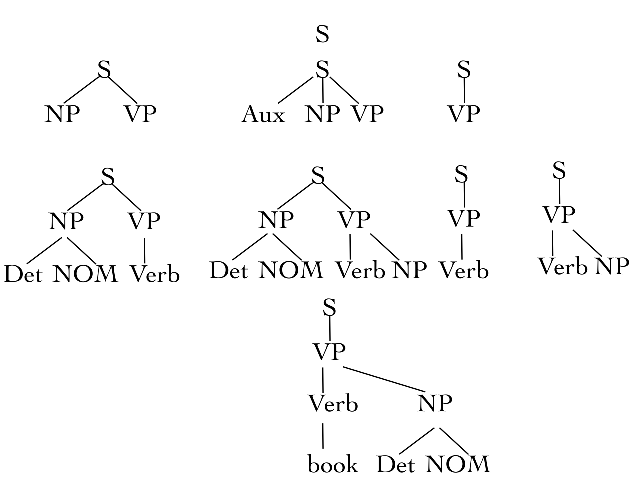
Top-Down Parsing

- Hypothesis-Driven Parsing
- Goal-directed
  - Start with S and get to string, w.
- Can use depth-first or breadth-first search

Top-Down Parsing

S → NP VP	VP → Verb
S → Aux NP VP	VP → Verb NP
S → VP	Det → that | this | a | the
NP → Det NOM	Noun → book | flight | meal | man
NOM → Noun	Verb → book | include | read
NOM → Noun NOM	Aux → does

Parse: *Book that flight*

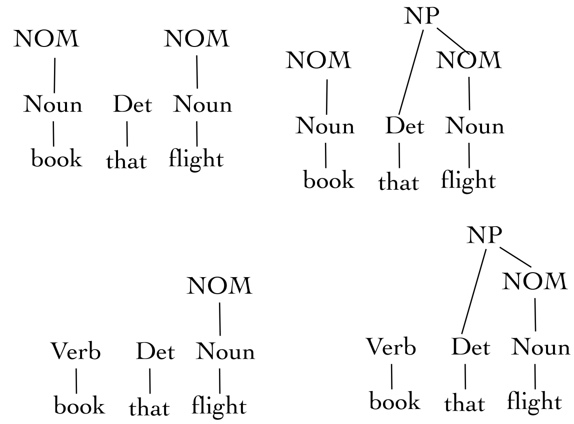


Probs w/Top-Down

- Left recursive rules, e.g. NP → NP PP lead to infinite recursion.
- Inefficient if lots of rules w/ same LHS
- Useless work: Expands trees that have no evidence.
- Do well if search directed by grammar.
- Treat pre-terminals before start parse.

Bottom-Up Parsing

- Data-driven: Start w/ string. Rewrite by replacing RHS by LHS of rules until get S.
- May have several RHS matches.
- Usually presented as shift-reduce parse



Shift-Reduce

sentence → NounPhrase VerbPhrase
NounPhrase → Art Noun
VerbPhrase → Verb | Adverb Verb
Art → the | a | ...
Verb → jumps | sings | ...
Noun → dog | cat | ...

Parse: *The dog jumps*

Draw trees as parse!

Stack	**Input Sequence**	
()	(the dog jumps)	
(the)	(dog jumps)	SHIFT word onto stack
(Art)	(dog jumps)	REDUCE using grammar rule
(Art dog)	(jumps)	SHIFT..
(Art Noun)	(jumps)	REDUCE..
(NounPhrase)	(jumps)	REDUCE
(NounPhrase jumps)	()	SHIFT
(NounPhrase Verb)	()	REDUCE
(NounPhrase VerbPhrase)	()	REDUCE
(Sentence)	()	SUCCESS

Bottom-Up Parsing

- Do we shift or reduce?
- If reduce, which rule do we use?
- With prog. langs, build table to always tell you what to do -- deterministic.
- Programming languages designed to be unambiguous. We don't have that luxury!
- ε-rules can be applied anywhere!
- May need to backtrack!

Top-Down vs. Bottom-Up

- Top-down may explore paths that can never result in desired string
  - In prog. langs, can make sure that doesn't happen.
- Bottom up may build subtrees that can not be part of trees rooted at S.
- Both may have to repeat work when backtracking!

Keys to Success

- Watch out for bad grammars
  - left-recursive for top-down (VP → VP PP)
- Try to avoid redoing work when backtracking
- Grammar transformations help
  - ... but linguists will hate you!

Dynamic Programming

- Next time:
  - CYK
  - Earley's Algorithm
  - Chart parsing
- Probabilistic versions

Any Questions?