Context-Free Grammar

CSCI-GA.2590 – Lecture 3

Ralph Grishman

A Grammar Formalism

- We have informally described the basic constructs of English grammar
- Now we want to introduce a formalism for representing these constructs
  - a formalism that we can use as input to a *parsing* procedure

Context-Free Grammar

- A context-free grammar consists of
  - a set of non-terminal symbols A, B, C, … ε N
  - a set of terminal symbols a, b, c, … ε T
  - a start symbol S ε N
  - a set of productions P of the form N→ (N U T)*

A Simple Context-Free Grammar

A simple CFG:

S → NP VP

NP → cats

NP → the cats

NP → the old cats

NP → mice

VP → sleep

VP → chase NP

Derivation and Language

If $A \rightarrow \beta$ is a production of the grammar, we can rewrite

$\alpha A \gamma \Rightarrow \alpha \beta \gamma$

A derivation is a sequence of rewrite operations

$\Rightarrow \ldots \Rightarrow \ldots \Rightarrow$

NP VP ➔ cats VP ➔ cats chase NP

The language generated by a CFG is the set of strings (sequences of terminals) which can be derived from the start symbol

$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow T^*$

S➔ NP VP ➔cats VP ➔ cats chase NP ➔ cats chase mice

Preterminals

It is convenient to include a set of symbols called *preterminals* (corresponding to the parts of speech) which can be directly rewritten as terminals (words)

This allows us to separate the productions into a set which generates sequences of preterminals (the "grammar") and those which rewrite the preterminals as terminals (the "dictionary")

A Grammar with Preterminals

grammar:

- S → NP VP
- NP → N
- NP → ART N
- NP → ART ADJ N
- VP → V
- VP → V NP

dictionary:

- N → cats
- N → mice
- ADJ → old
- DET → the
- V → sleep
- V → chase

Grouping Alternates

- To make the grammar more compact, we group productions with the same left-hand side:

  S → NP VP

  NP → N | ART N | ART ADJ N

  VP → V | V NP

- A grammar can be used to
  - generate
  - recognize
  - parse

- Why parse?
  - parsing assigns the sentence a structure that may be helpful in determining its meaning

vs Finite State Language

- CFGs are more powerful than finite-state grammars (regular expressions)
  - FSG cannot generate center embeddings

$$S \rightarrow (S) \mid x$$

  - even if FSG can capture the language, it may be unable to assign the nested structures we want

A slightly bigger CFG

sentence → np vp

np → ngroup | ngroup pp

ngroup → n | art n | art adj n

vp → v | v np | v vp | v np pp (auxilliary)

pp → p np (pp = prepositional phrase)

Ambiguity

- Most sentences will have more than one parse
- Generally different parses will reflect different meanings …

  “I saw the man with a telescope.”

  Can attach pp (“with a telescope”) under np or vp

A CFG with just 2 nonterminals

S → NP V | NP V NP

NP → N | ART NOUN | ART ADJ N

use this for tracing our parsers

Top-down parser

repeat

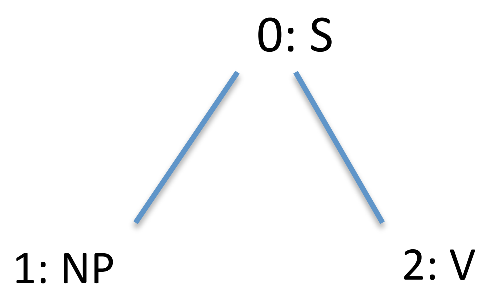
- expand leftmost non-terminal using first production (save any alternative productions on backtrack stack)
- if we have matched entire sentence, quit (success)
- if we have generated a terminal which doesn't match sentence, pop choice point from stack (if stack is empty, quit (failure))

Top-down parser

0: S

the cat chases mice

Top-down parser

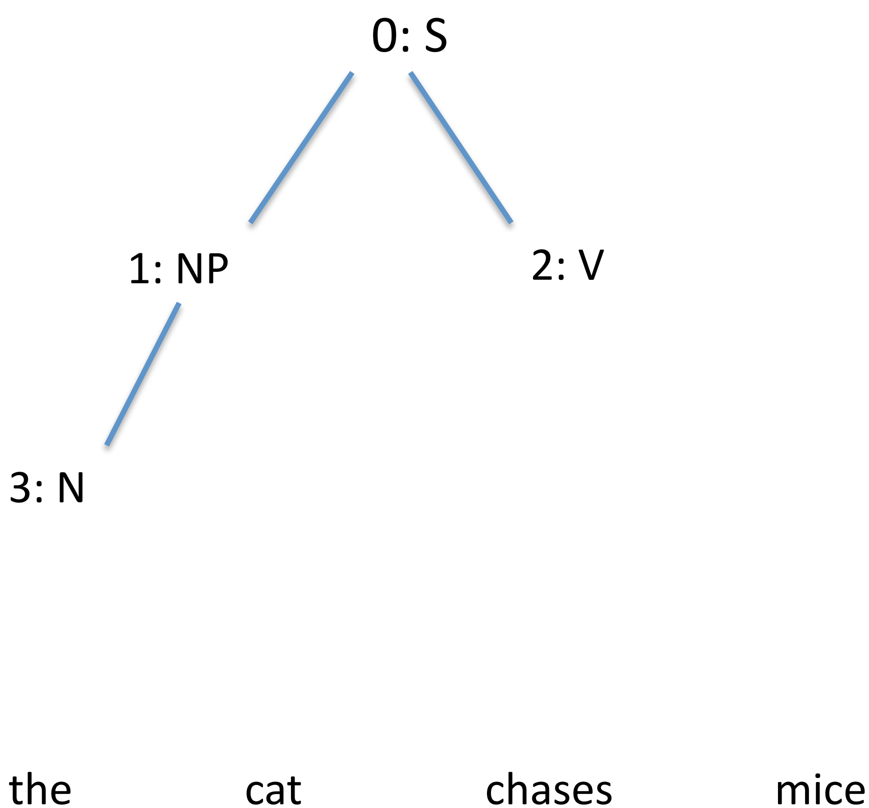


backtrack table

0: S → NP V NP

the cat chases mice

Top-down parser



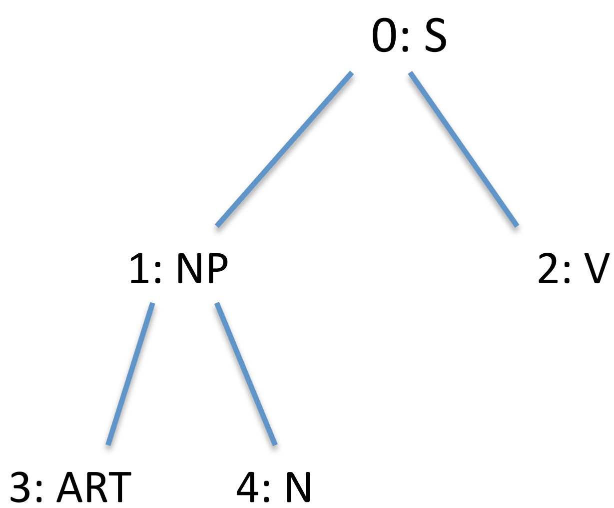
backtrack table

0: S → NP V NP

1:NP → ART ADJ N

1: NP → ART N

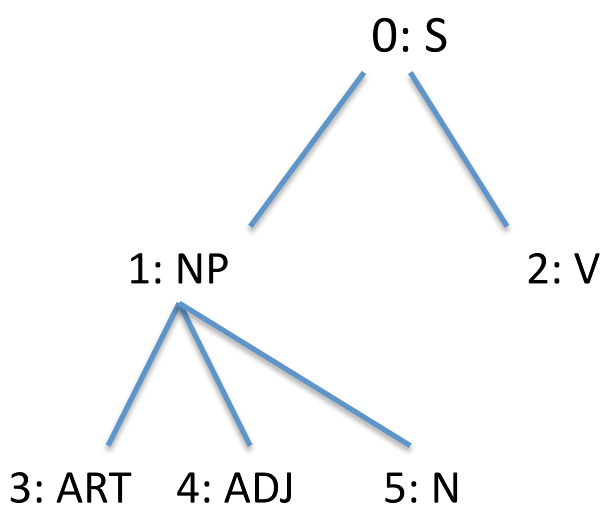
Top-down parser



backtrack table
0: S → NP V NP
1:NP → ART ADJ N

the cat chases mice

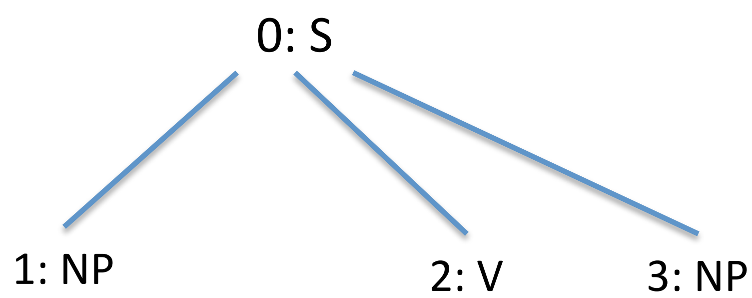
Top-down parser



backtrack table
0: S → NP V NP

the cat chases mice

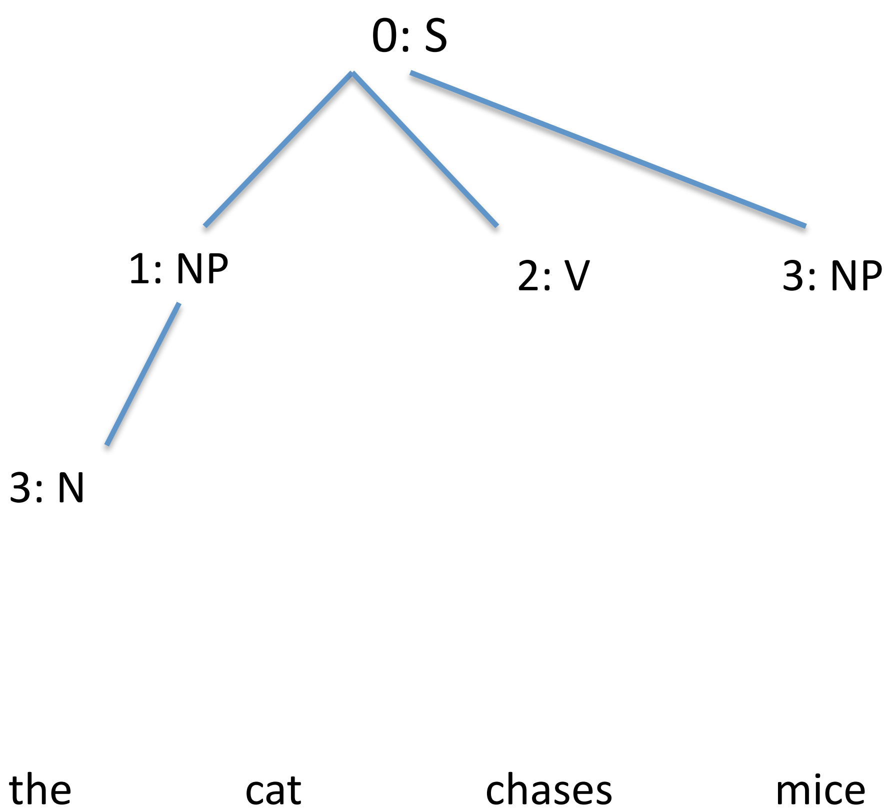
Top-down parser



backtrack table

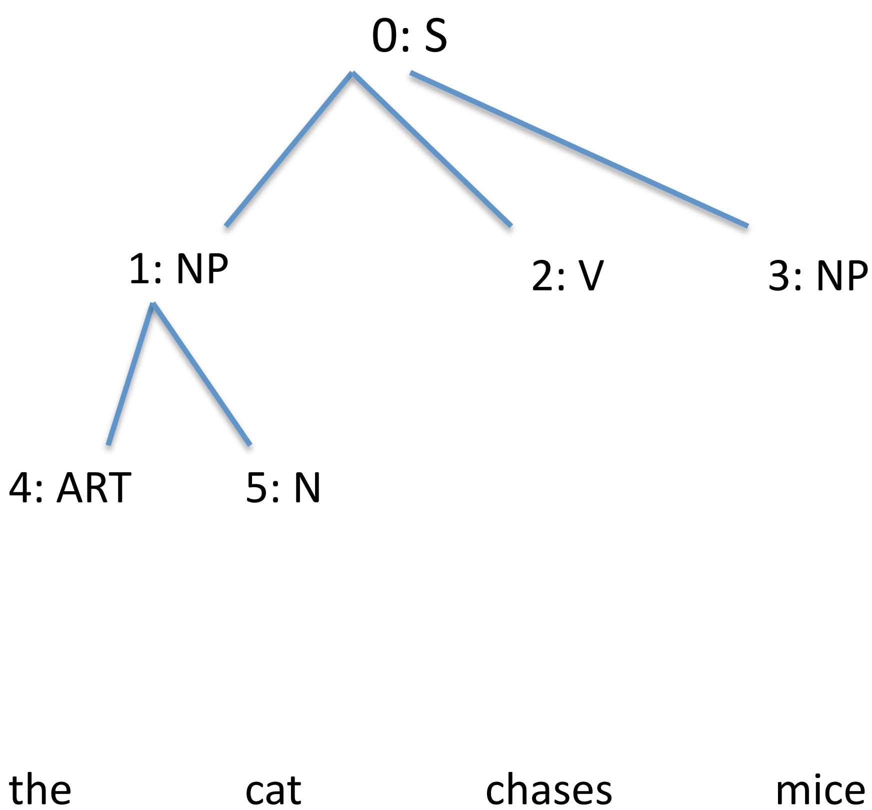
the cat chases mice

Top-down parser



backtrack table
1:NP → ART ADJ N
1: NP → ART N

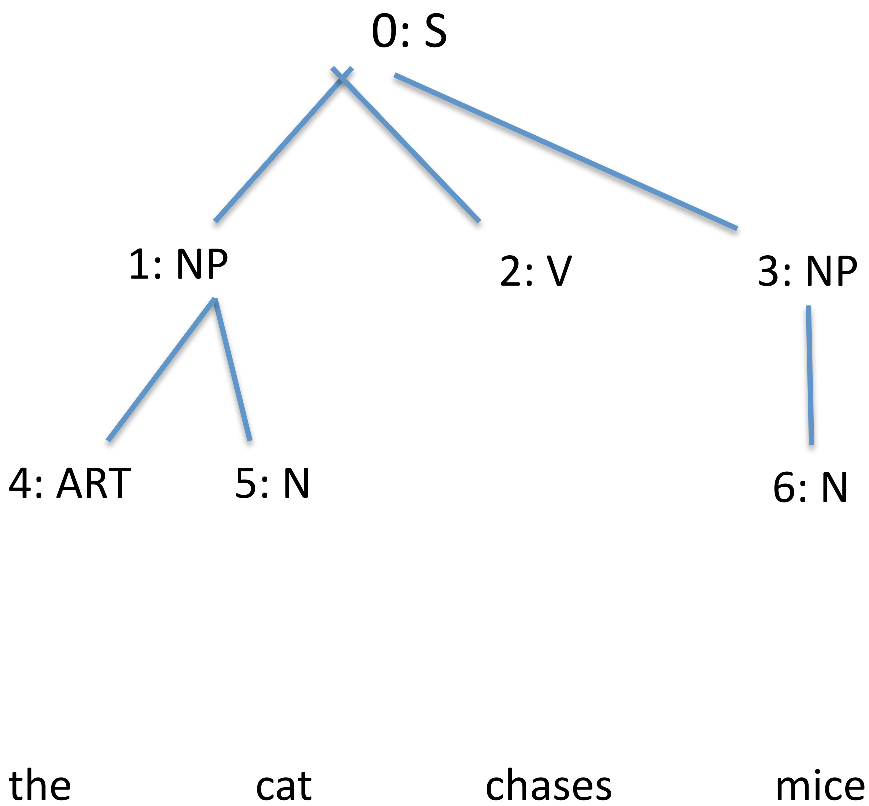
Top-down parser



backtrack table

1:NP → ART ADJ N

Top-down parser



backtrack table

1:NP → ART ADJ N

3: NP → ART ADJ N

3: NP → ART N

parse!

Bottom-up parser

- Builds a table

symbol	start	end	constituents
N	0	1	-

where each row represents a parse tree node spanning the words from *start* up to *end*

Bottom-up parser

- We initialize the table with the parts-of – speech of each word …

symbol	start	end	constituents
ART	0	1	-
N	1	2	-
V	2	3	-
N	3	4	-

Bottom-up parser

- We initialize the table with the parts-of – speech of each word …

symbol	start	end	constituents
ART	0	1	-
N	1	2	-
V	2	3	-
N	2	3	-
N	3	4	-

- remembering that many English words have several parts of speech

Bottom-up parser

- Then if these is a production A→B C and we have entries for B and C with $end_B = start_C$, we add an entry for A with $start = start_B$ and $end = end_C$

node #	symbol	start	end	constituents
0	ART	0	1	-
1	N	1	2	-
2	V	2	3	-
3	N	2	3	-
4	N	3	4	-
5	NP	0	2	[0, 1]

[see lecture notes for handling general productions]

Bottom-up parser

node #	symbol	start	end	constituents
0	ART	0	1	-
1	N	1	2	-
2	V	2	3	-
3	N	2	3	-
4	N	3	4	-
5	NP	0	2	[0, 1]
6	NP	1	2	[1]
7	NP	2	3	[3]
8	NP	3	4	[4]
9	S	0	4	[5, 2, 8] ← parse!
10	S	1	4	[6, 2, 8]

several more S's