Introduction to NLP

Parsing algorithms

Alexandre Rademaker[1]

FGV/EMAp

May 9, 2018

[1] Olga Zamaraeva

Overview

- We will focus on the concept of parsing
- and why naive parsing is inefficient
- Dynamic programming idea
- Then will briefly cover CKY
- and then an assignment will ask you to study it in detail
- we discussed probabilistic parsing

Parsing

- Recognizing string as input and assigning structure to it
- Syntactic parsing: assigning syntactic structure
- Semantic parsing: assigning semantic structure

Syntactic Parsing

Parsing: Making explicit structure that is inherent (implicit) in natural language strings

- What is that structure?
- Why would we need it?

Bottom Up vs. Top Down parsing

```
        S
       /|\

       ???

   |  |  |  |  |  |
   a  b  c  d  d  f
```

Recursive Top-Down Parsing

A Top-Down Parser

Input: CFG grammar, lexicon, sentence to parse

Output: yes/no

State of the parse: (*symbol list*, *position*)

$${}_1 \text{ The } {}_2 \text{ old } {}_3 \text{ man } {}_4 \text{ cried } {}_5$$

start state: ((S) 1)

slide from: https://www.cs.cornell.edu/courses/cs4740/2012sp/lectures/parsing-intro-4pp.pdf

Recursive Top-Down Parsing

Grammar and Lexicon

Grammar:

1. S $\rightarrow$ NP VP
2. NP $\rightarrow$ art n
3. NP $\rightarrow$ art adj n
4. VP $\rightarrow$ v
5. VP $\rightarrow$ v NP

Lexicon:

the: art

old: adj, n

man: n, v

cried: v

$$_1 \text{ The } _2 \text{ old } _3 \text{ man } _4 \text{ cried } _5$$

slide from: `https://www.cs.cornell.edu/courses/cs4740/2012sp/lectures/parsing-intro-4pp.pdf`

Recursive Top-Down Parsing

Algorithm for a Top-Down Parser

$PSL \leftarrow$ (((S) 1))

1. *Check for failure.* If PSL is empty, return NO.
2. *Select the current state, C.* C $\leftarrow$ pop (PSL).
3. *Check for success.* If C = (() <final-position>), YES.
4. *Otherwise, generate the next possible states.*
   (a) $s_1 \leftarrow$ first-symbol(C)
   (b) If s_1 is a *lexical symbol* and next word can be in that class, create new state by removing s_1, updating the word position, and adding it to PSL. (I'll add to front.)
   (c) If s_1 is a *non-terminal*, generate a new state for each rule in the grammar that can rewrite s_1. Add all to PSL. (Add to front.)

slide from: `https://www.cs.cornell.edu/courses/cs4740/2012sp/lectures/parsing-intro-4pp.pdf`

Recursive Top-Down Parsing

Example

Current state	Backup states	
1. ((S) 1)		
2. ((NP VP) 1)		
3. ((art n VP) 1)	((art adj n VP) 1)	
4. ((n VP) 2)	((art adj n VP) 1)	
5. ((VP) 3)	((art adj n VP) 1)	
6. ((v) 3)	((v NP) 3) ((art adj n VP) 1)	
7. (() 4)	((v NP) 3) ((art adj n VP) 1)	Backtrack

slide from: `https://www.cs.cornell.edu/courses/cs4740/2012sp/lectures/parsing-intro-4pp.pdf`

Recursive Top-Down Parsing

8. ((v NP) 3)	((art adj n VP) 1)	leads to backtracking
...		
9. ((art adj n VP) 1)		
10. ((adj n VP) 2)		
11. ((n VP) 3)		
12. ((VP) 4)		
13. ((v) 4)	((v NP) 4)	
14. (() 5)	((v NP) 4)	
YES		

DONE!

slide from: https://www.cs.cornell.edu/courses/cs4740/2012sp/lectures/parsing-intro-4pp.pdf

Recursive Top-Down Parsing

- What is the time complexity of what we just saw?

Recursive Top-Down Parsing

- What is the time complexity of what we just saw?
  - **Exponential** in n!
  - (Meaning, as we increase the length of input, the time to do parsing increases **exponentially**)
  - (Which is very very bad)

**Optional exercise: Recursive Parsing Running Time

S $\rightarrow$ A
A $\rightarrow$ a A b | a A c | ϵ

- Consider a string $a^n c^n$
- Convince yourself that you will expand A 2^n times

Example

Grammar	Lexicon
$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid the \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Pronoun$	$Pronoun \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid NWA$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	$Preposition \rightarrow from \mid to \mid on \mid near \mid through$
$Nominal \rightarrow Nominal\ Noun$	
$Nominal \rightarrow Nominal\ PP$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	
$VP \rightarrow Verb\ NP\ PP$	
$VP \rightarrow Verb\ PP$	
$VP \rightarrow VP\ PP$	
$PP \rightarrow Preposition\ NP$	

Figure 12.1 The $\mathscr{L}_1$ miniature English grammar and lexicon.

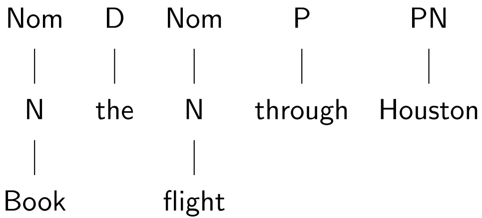
Bottom-up parsing

Book the flight through Houston

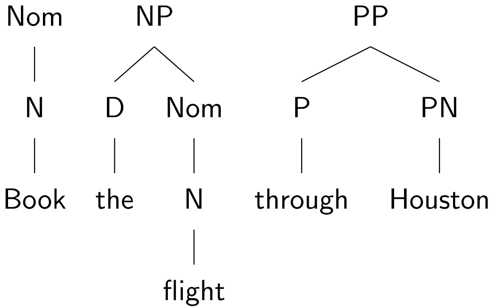
Bottom-up parsing

N	D	N	P	PN
Book	the	flight	through	Houston

Bottom-up parsing

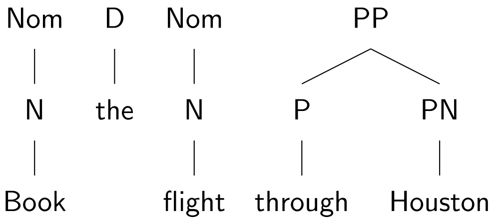


Bottom-up parsing

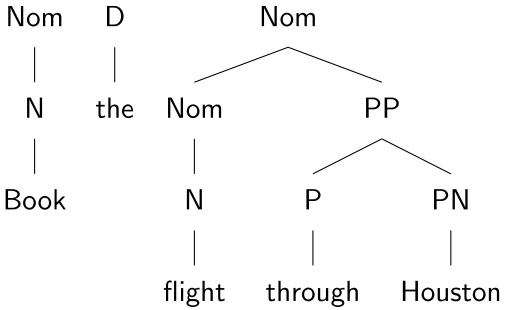


No more possibilities! Backtrack...

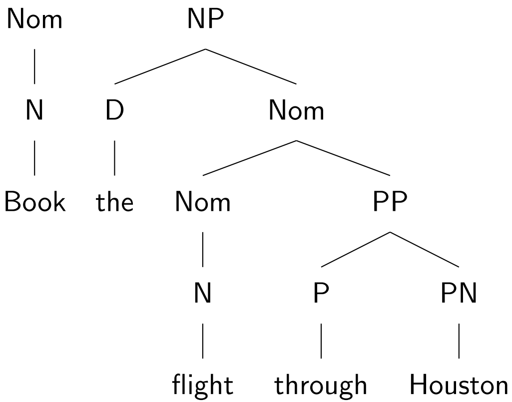
Bottom-up parsing



Bottom-up parsing



Bottom-up parsing



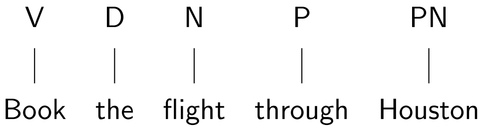
No more possibilities! Backtrack... Up to where?..

Bottom-up parsing

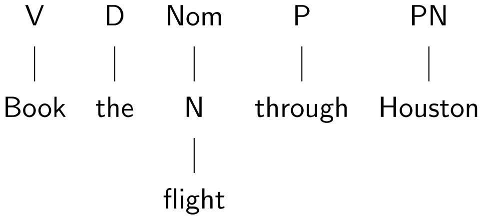
Backtrack to the very beginning, actually!

Book the flight through Houston

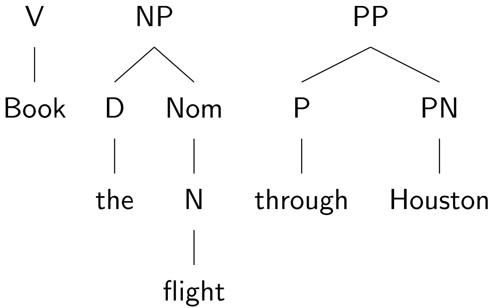
Bottom-up parsing



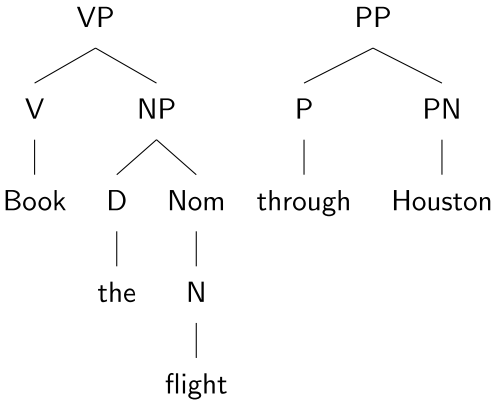
Bottom-up parsing



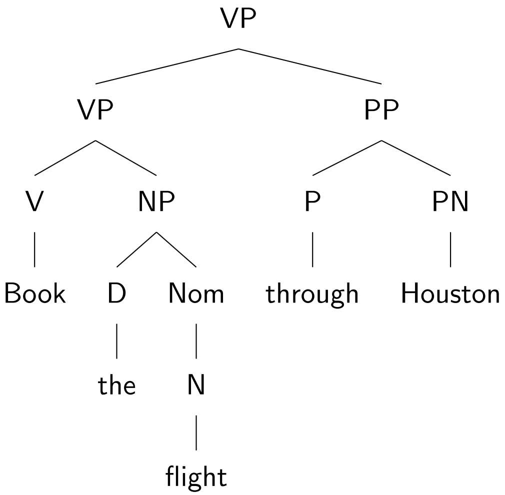
Bottom-up parsing



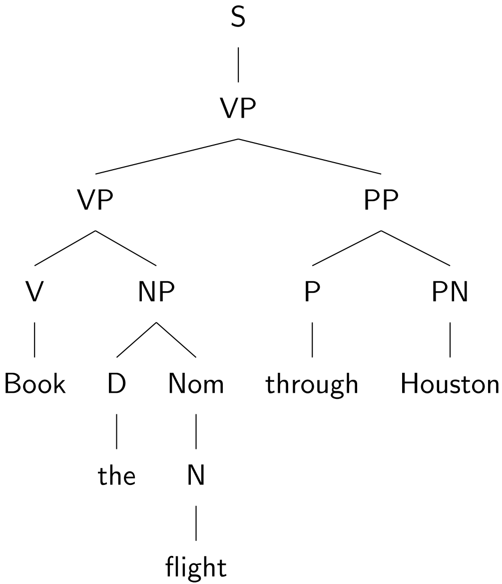
Bottom-up parsing



Bottom-up parsing

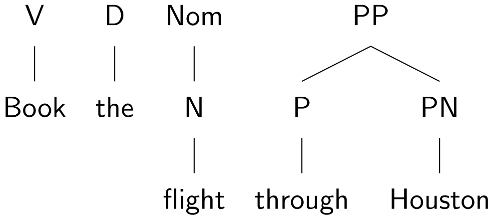


Bottom-up parsing

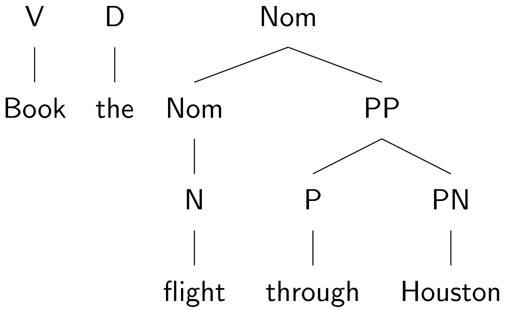


Bottom-up parsing

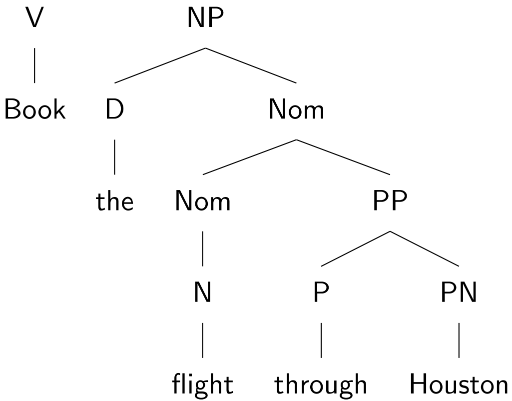
Or, we could have instead done:



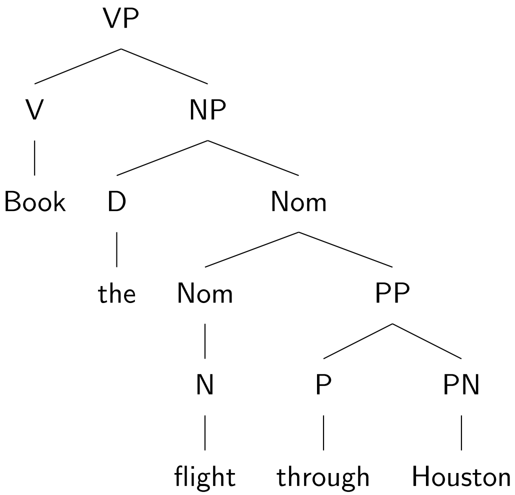
Bottom-up parsing



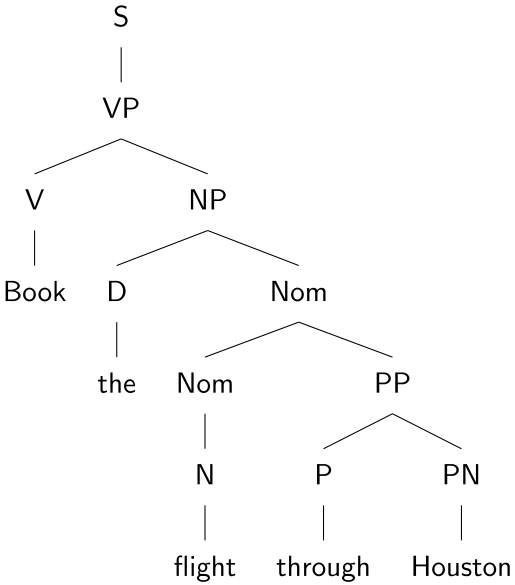
Bottom-up parsing



Bottom-up parsing



Bottom-up parsing



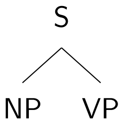
How do we make sure we get both trees?

- Go through **all** possibilities for productions

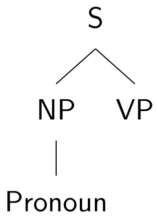
Top-Down Parsing

S

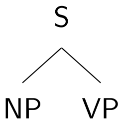
Top-Down Parsing



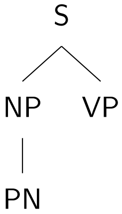
Top-Down Parsing



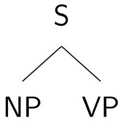
Top-Down Parsing



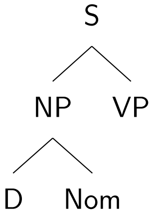
Top-Down Parsing



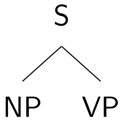
Top-Down Parsing



Top-Down Parsing



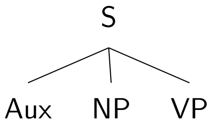
Top-Down Parsing



Top-Down Parsing

S

Top-Down Parsing



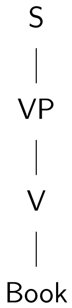
Top-Down Parsing

S

Top-Down Parsing

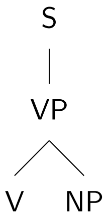
Top-Down Parsing

Top-Down Parsing

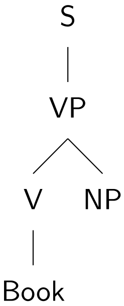


Yes, but we have more input still...

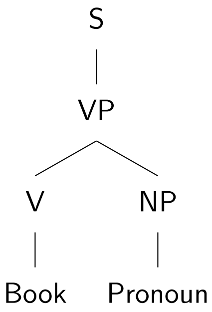
Top-Down Parsing



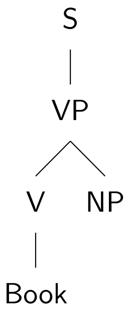
Top-Down Parsing



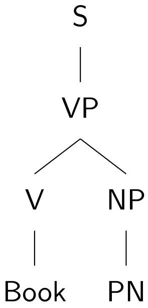
Top-Down Parsing



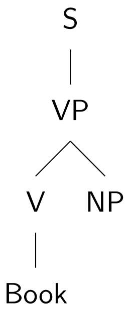
Top-Down Parsing



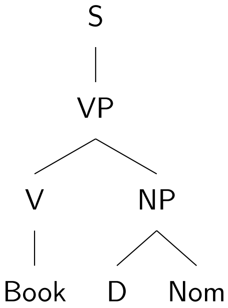
Top-Down Parsing



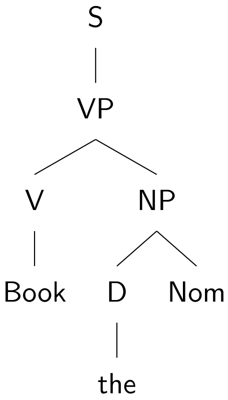
Top-Down Parsing



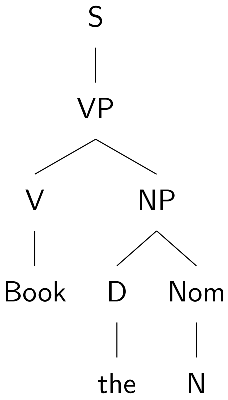
Top-Down Parsing



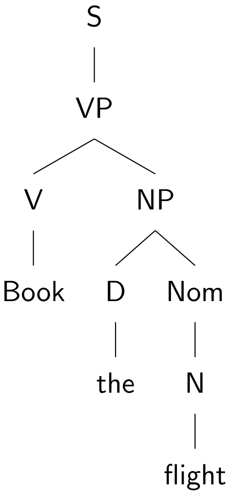
Top-Down Parsing



Top-Down Parsing

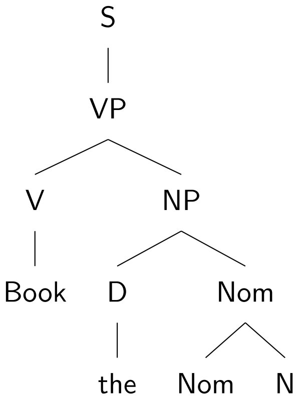


Top-Down Parsing

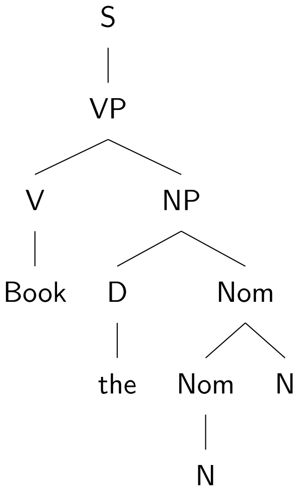


Yes, but we have more input still...

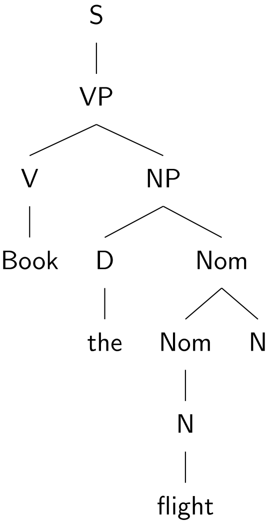
Top-Down Parsing



Top-Down Parsing

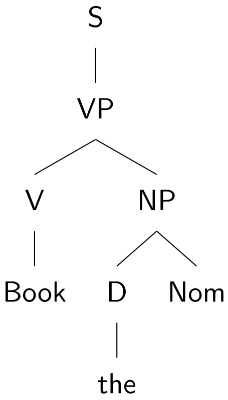


Top-Down Parsing

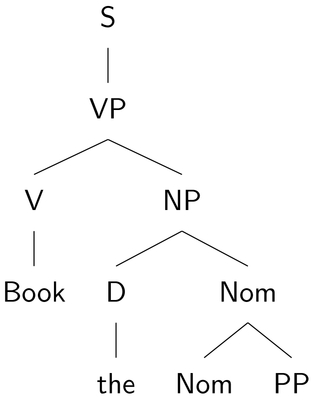


Nope... Backtrack again...

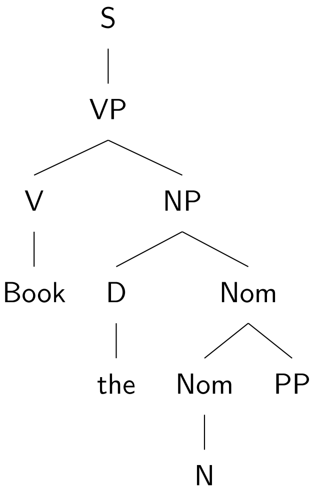
Top-Down Parsing



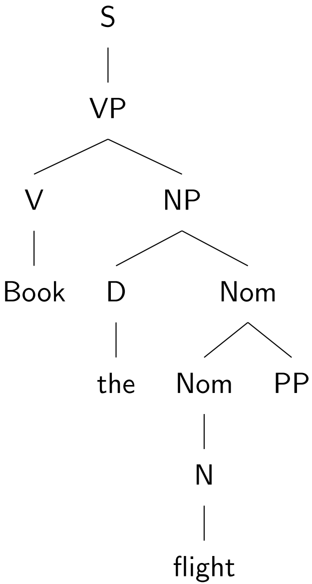
Top-Down Parsing



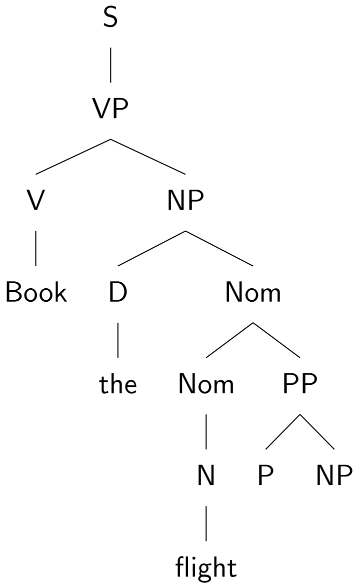
Top-Down Parsing



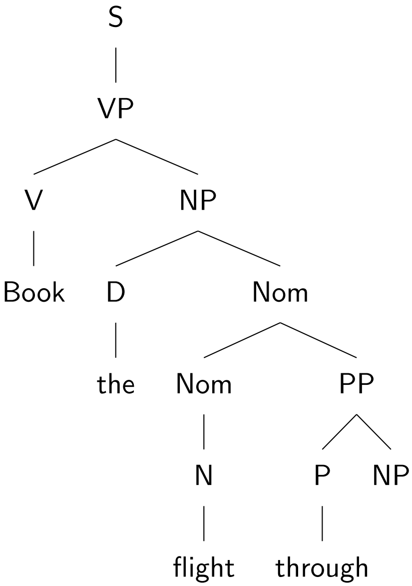
Top-Down Parsing



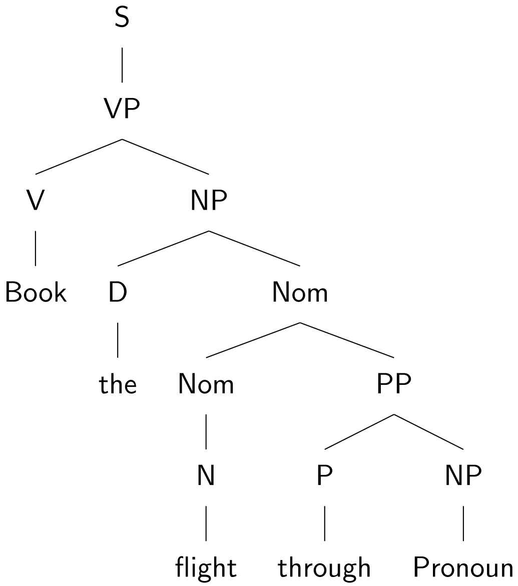
Top-Down Parsing



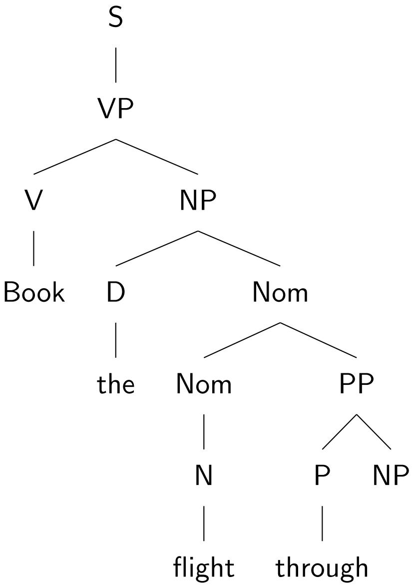
Top-Down Parsing



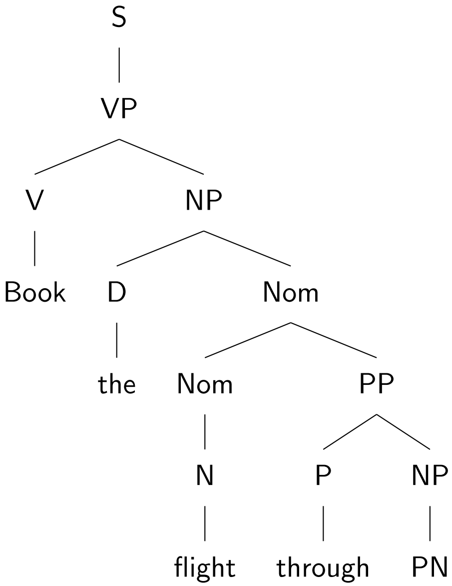
Top-Down Parsing



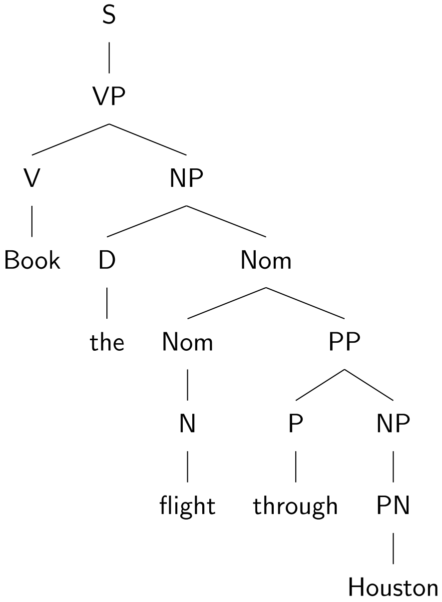
Top-Down Parsing



Top-Down Parsing



Top-Down Parsing



Top-Down Parsing

- Could we have gotten the second tree by top-down parsing?

Top-Down Parsing

- Could we have gotten the second tree by top-down parsing?
  - Yes; it is a matter of which rule happened to be on the top of the stack
  - We grabbed **VP** $\rightarrow$ **V NP**
  - But the option **VP** $\rightarrow$ **VP PP** is also on the stack somewhere
  - Thus the returned parse is subject to an arbitrary listing of rules in the grammar

Bottom Up vs. Top Down parsing

- Top-down parsers do not waste time exploring hypotheses not leading to S
  - ...but do waste time exploring hypotheses not matching the input
- Bottom-up parsers do not waste time exploring hypotheses not matching input
  - ...but do waste time exploring hypotheses not leading to S
- Both can take exponential time
  - (in the worst case, easier shown on abstract CFG)
  - Some recursive parsers are $O(n^4)$
- An answer to poor time complexity: **dynamic programming**
  - $O(n^3)$

Example: The Fibonacci numbers

Recursive definition: f(0) = 0; f(1) = 1 f(n) = f(n-1) + f(n-2)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657...

f(100) = 218922995834555169026

The Fibonacci numbers: naive implementation

- Since we have a recursive definition, let's implement the Fibonacci numbers printer recursively!

```
def fibonacci(n):
    if n in [0,1]:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

What's the problem with this?

The Fibonacci numbers: better implementation

```
def fibonacci(n):
    return fibonacci_helper(n,{})

def fibonacci_helper(n,memo):
    if n in [0,1]:
        return n
    if not n in memo:
        memo[n] = fibonacci_helper(n-1,memo)
            + fibonacci_helper(n-2,memo)
    return memo[n]
```

Dynamic programming

- Fill in a table with solutions to subproblems
- Then can just look up momentarily the precomputed solution
- No need to perform the same computation many times

Fibonacci numbers

- $f(0) = 0$
- $f(1) = 1$
- $f(2) = f(1) + f(0)$
  - what's $f(1)$?
  - what's $f(0)$?

Fibonacci numbers

- $f(3) = f(2) + f(1)$
  - $f(2) = f(1) + f(0)$
  - $f(1) = 1$
  - $f(0) = 0$

Fibonacci numbers

- $f(4) = f(3) + f(2)$
  - $f(3) = f(2) + f(1)$
  - $f(2) = f(1) + f(0)$
  - $f(1) = 1$
  - $f(0) = 0$

Fibonacci numbers

- $f(5) = f(4) + f(3)$
  - $f(4) = f(3) + f(2)$
  - $f(3) = f(2) + f(1)$
  - $f(2) = f(1) + f(0)$
  - $f(1) = 1$
  - $f(0) = 0$
- etc... (deep recursion; slow; do the same computation again and again)

Fibonacci numbers

$f(0) = ?$

	0	1	2	3	4	5

Fibonacci numbers

f(0) = ?

Not in the table, so compute: f(0)=0 (or rather, return the base case)

fill in the cell

	0	1	2	3	4	5

Fibonacci numbers

$f(1) = ?$

Not in the table, so compute: f(1)=1 (or rather, return the base case)

fill in the cell

	0	1	2	3	4	5
	0					

Fibonacci numbers

f(2) = ?

Not in the table, so compute: f(2)=f(2-1) + f(2-2) = f(1) + f(0)

But both f(1) and f(0) are already in the table! No need to compute! Just look up!

fill in the cell

	0	1	2	3	4	5
	0	1				

Fibonacci numbers

f(3) = ?

Not in the table, so compute: f(3)=f(3-1) + f(3-2) = f(2) + f(1)

But both f(2) and f(1) are already in the table! No need to compute! Just look up!

fill in the cell

	0	1	2	3	4	5
	0	1	1			

Fibonacci numbers

f(4) = ?

Not in the table, so compute: f(4)=f(4-1) + f(4-2) = f(3) + f(2)

But both f(3) and f(2) are already in the table! No need to compute! Just look up!

fill in the cell

	0	1	2	3	4	5
	0	1	1	2		

Dynamic programming for parsing

Once the constituent has been discovered, store the information

- Example: The CKY algorithm (Cocke-Kazami-Younger)

Chomsky Normal Form

- All productions must conform to two forms:
  - A $\rightarrow$ BC
  - A $\rightarrow$ *w*
- i.e. only binary branching trees (and leaves)

Chomsky Normal Form

- To convert to CNF:
  - copy all conforming rules as is
  - Flatten unit productions
    - **Nom** $\rightarrow$ **N**, **N** $\rightarrow$ *cat* | *dog* becomes: **Nom** $\rightarrow$ *cat* , **Nom** $\rightarrow$ *dog*
  - Introduce "dummy" rules to get rid of mixed terminal-nonterminal RHS (right-hand side)
    - **INF** $\rightarrow$ *to* **VP** becomes: **TO** $\rightarrow$ *to* and **INF** $\rightarrow$ **TO VP**
  - Introduce "dummy" rules to expand rules with RHS greater than 2 nonterminals
    - **A** $\rightarrow$ **B C D** becomes: **A** $\rightarrow$ **X D**, X $\rightarrow$ **B C**

Original grammar

Grammar	Lexicon
$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid the \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal \mid money$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid prefer$
$NP \rightarrow Pronoun$	$Pronoun \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$Proper\text{-}Noun \rightarrow Houston \mid NWA$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$Nominal \rightarrow Noun$	$Preposition \rightarrow from \mid to \mid on \mid near \mid through$
$Nominal \rightarrow Nominal\ Noun$	
$Nominal \rightarrow Nominal\ PP$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb\ NP$	
$VP \rightarrow Verb\ NP\ PP$	
$VP \rightarrow Verb\ PP$	
$VP \rightarrow VP\ PP$	
$PP \rightarrow Preposition\ NP$	

Figure 12.1 The $\mathscr{L}_1$ miniature English grammar and lexicon.

Chomsky Normal Form

$\mathscr{L}_1$ **Grammar**	$\mathscr{L}_1$ **in CNF**
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow TWA \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

Figure 12.3 $\mathscr{L}_1$ Grammar and its conversion to CNF. Note that although they aren't shown

CKY: Main idea

- A 2-dimensional array (aka a table) can encode the structure of the tree
- Each cell [i,j] contains all constituents that span positions *i* through *j* of the input string
  - ${}_0 Book_1 that_2 flight_3$

  Cell [0,n] must have the Start symbol if we have a parse
  - ...and can have more than one!

CKY

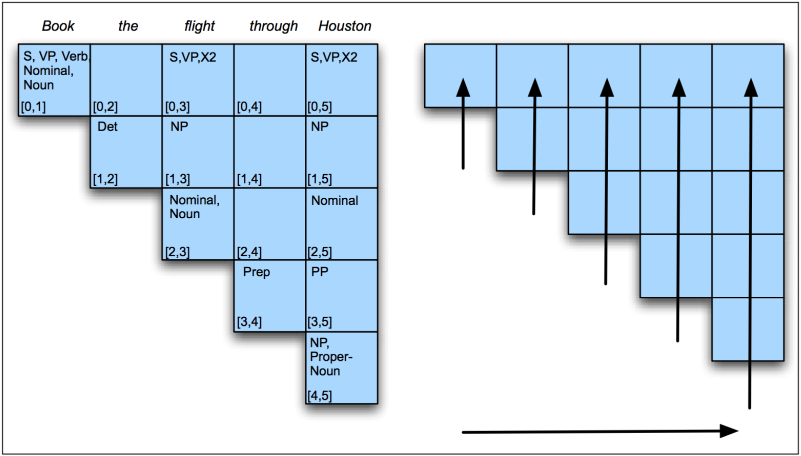
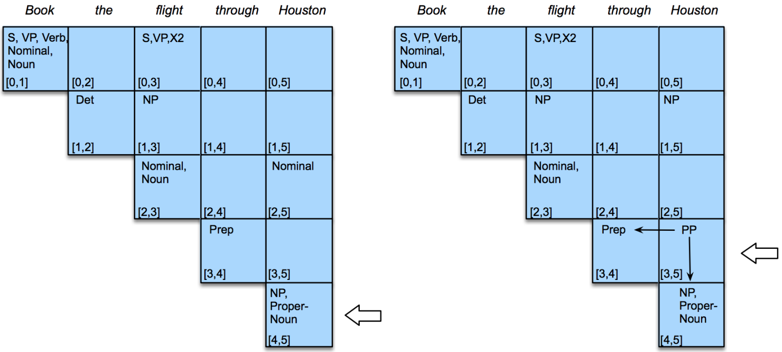
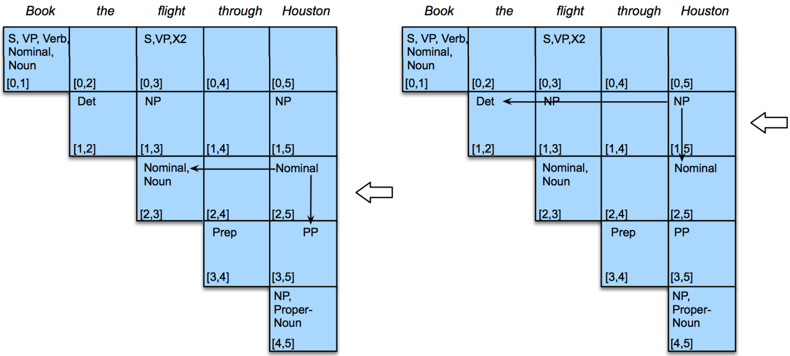


Figure 12.4 Completed parse table for *Book the flight through Houston.*

CKY

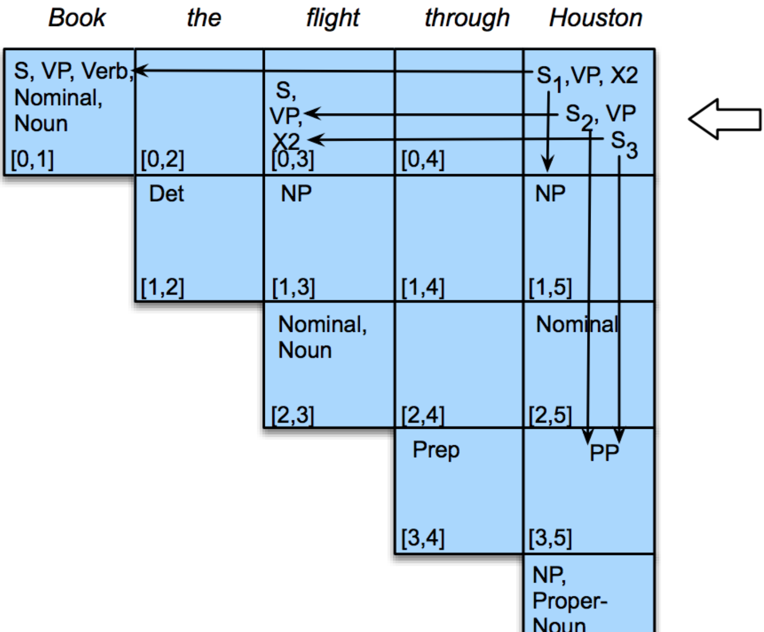

Book the flight through Houston
S, VP, Verb, Nominal, Noun [0,1]
[0,2]
S,VP,X2 [0,3]
[0,4]
[0,5]
Det [1,2]
NP [1,3]
[1,4]
[1,5]
Nominal, Noun [2,3]
[2,4]
Nominal [2,5]
Prep [3,4]
[3,5]
NP, Proper-Noun [4,5]
Book the flight through Houston
S, VP, Verb, Nominal, Noun [0,1]
[0,2]
S,VP,X2 [0,3]
[0,4]
[0,5]
Det [1,2]
NP [1,3]
[1,4]
NP [1,5]
Nominal, Noun [2,3]
[2,4]
[2,5]
Prep [3,4]
PP [3,5]
NP, Proper-Noun [4,5]

CKY



Book
the
flight
through
Houston
S, VP, Verb, Nominal, Noun
[0,1]
[0,2]
S,VP,X2
[0,3]
[0,4]
[0,5]
Det
[1,2]
NP
[1,3]
[1,4]
NP
[1,5]
Nominal, Noun
[2,3]
[2,4]
Nominal
[2,5]
Prep
[3,4]
PP
[3,5]
NP, Proper-Noun
[4,5]

CKY



Book
the
flight
through
Houston
S, VP, Verb, Nominal, Noun
[0,1]
[0,2]
S, VP, X2
[0,3]
[0,4]
S1,VP, X2
S2, VP
S3
Det
[1,2]
NP
[1,3]
[1,4]
NP
[1,5]
Nominal, Noun
[2,3]
[2,4]
Nominal
[2,5]
Prep
[3,4]
PP
[3,5]
NP, Proper-Noun

Limitations of classical CKY

- It is a recognizer
- Turn a recognizer into a parser by storing all tree paths leading to S
- ...but returning all possible trees is again exponential time!
- Also, we modified the grammar!

Limitations of classical CKY: Solutions

- Probabilistic parsing
- Train a probabilistic grammar and then return the most probable parse
- Modify CKY to be able to recover original grammar
- Employ e.g. partial parsing to get accommodate CFG directly (not in CNF)