CS114 Lecture 10
Parsing

March 5, 2014

Professor Meteer

Thanks for Jurafsky & Martin & Prof. Pustejovksy for slides

Announcements

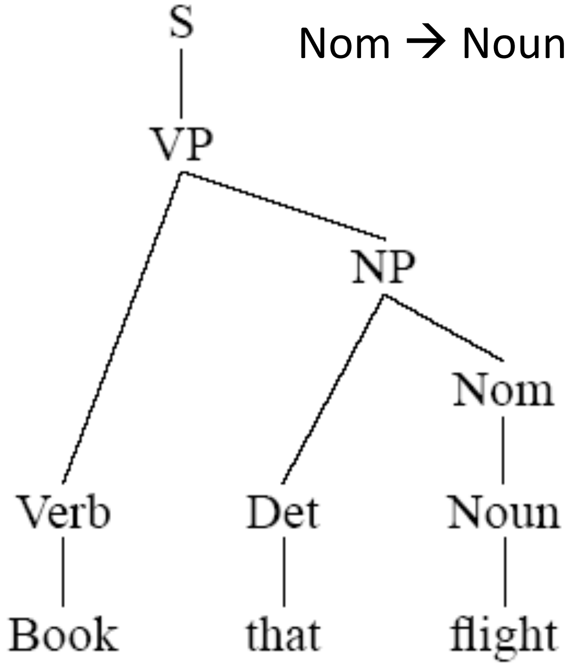
- Industry Meet and Greet
  - Tuesday March 11
- JBS: Summer 2014

PARSING

- Parsing is the process of recognizing and assigning STRUCTURE
- Parsing a string with a CFG:
  - Finding a derivation of the string consistent with the grammar
  - The derivation gives us a PARSE TREE

Grammar:

S → NP VP
S→ Aux NP VP
S → VP
NP → Det Nom
Nom → Noun
Nom → Noun Nom



PARSING AS SEARCH

- The main problem with parsing is the existence of CHOICE POINTS
- Parsing Strategy
  - Top down:
    - Expectation Driven
    - Start with “S”
  - Bottom up:
    - Data Driven
    - Start with words/categories
- Search Strategy
  - Determining the order alternatives are considered
    - Depth first
    - Breadth first

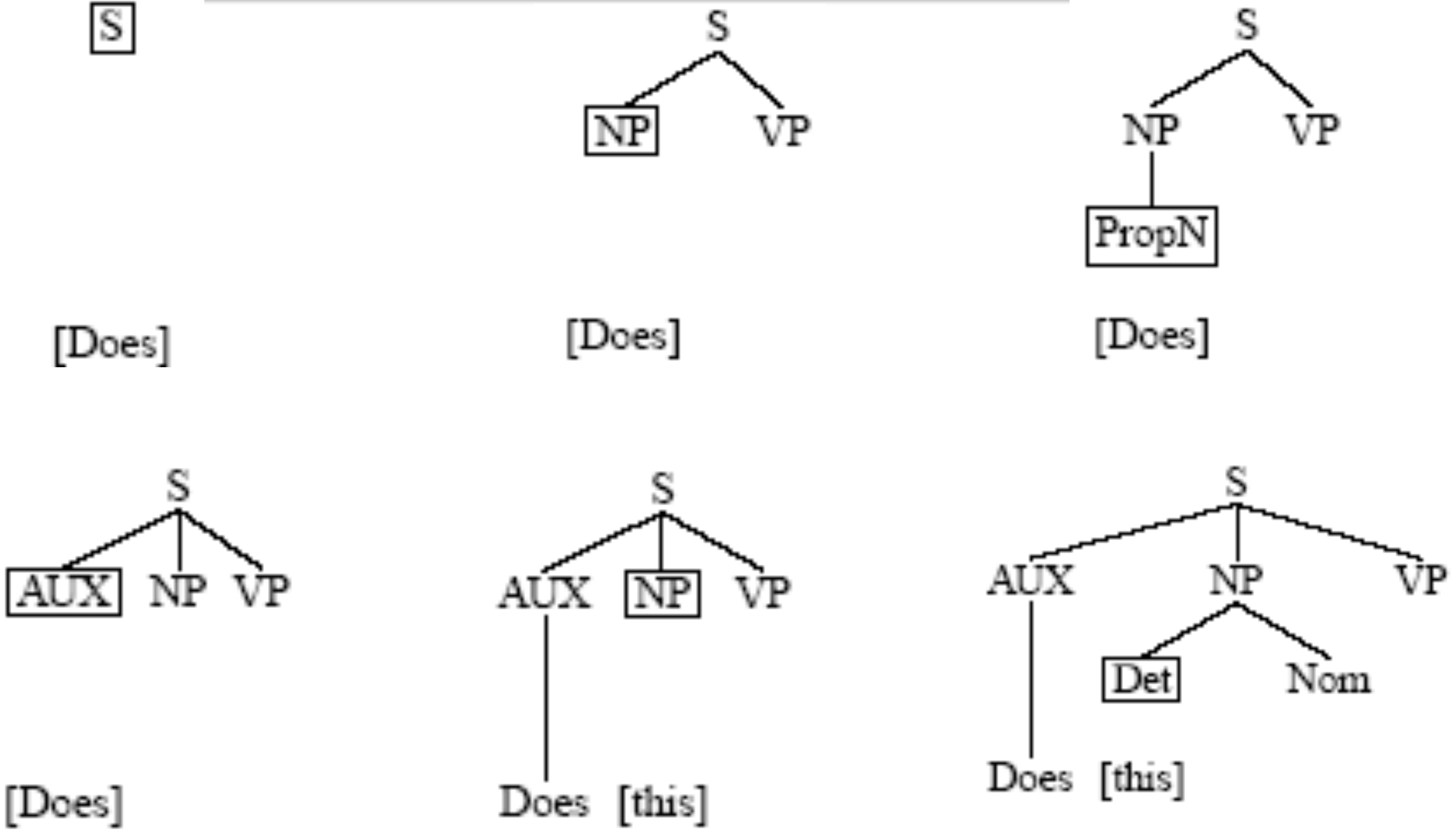
TOP-DOWN vs BOTTOM-UP

- TOP-DOWN:
  - Only search among grammatical answers
  - BUT: suggests hypotheses that may not be consistent with data
  - Problem: left-recursion
- BOTTOM-UP:
  - Only forms hypotheses consistent with data
  - BUT: may suggest hypotheses that make no sense globally

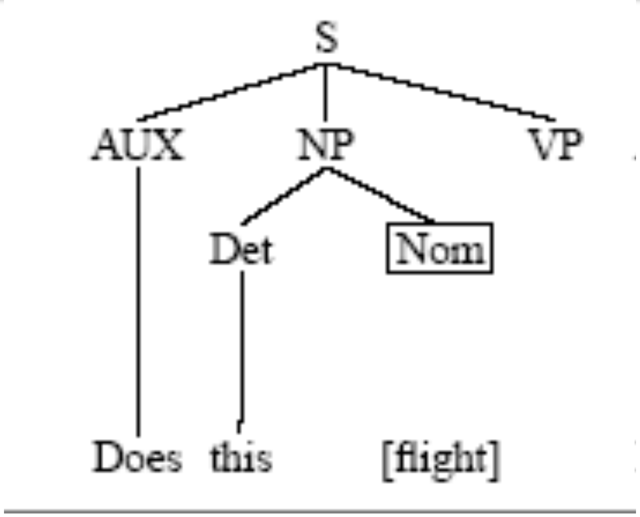
NON-PARALLEL SEARCH

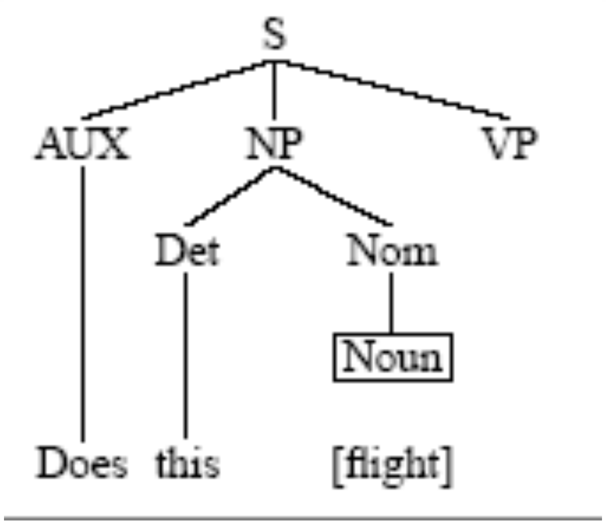
- If it's not possible to examine all alternatives in parallel, it's necessary to make further decisions:
  - Which node in the current search space to expand first (breadth-first or depth-first)
  - Which of the applicable grammar rules to expand first
  - Which leaf node in a parse tree to expand next (e.g., leftmost)

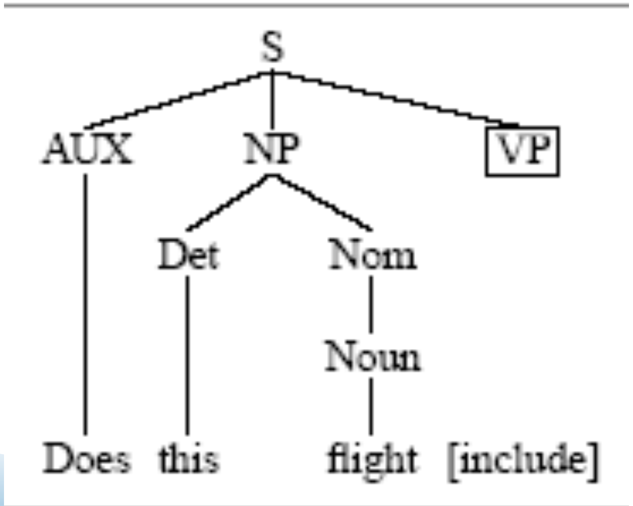
TOP-DOWN, DEPTH-FIRST, LEFT-TO-RIGHT

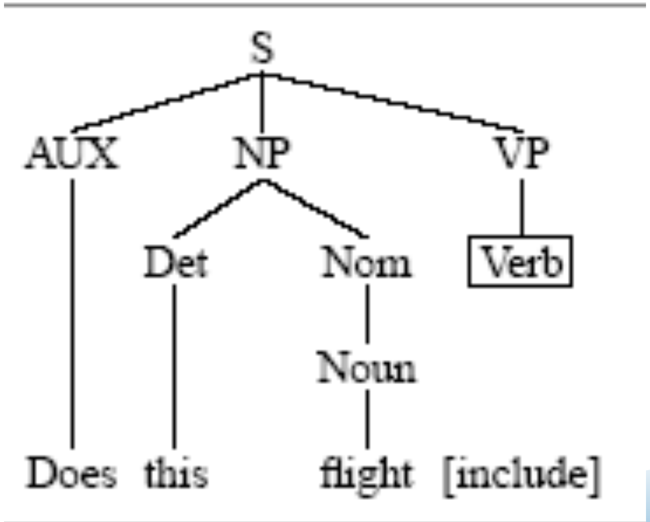


TOP-DOWN, DEPTH-FIRST, LEFT-TO-RIGHT (II)

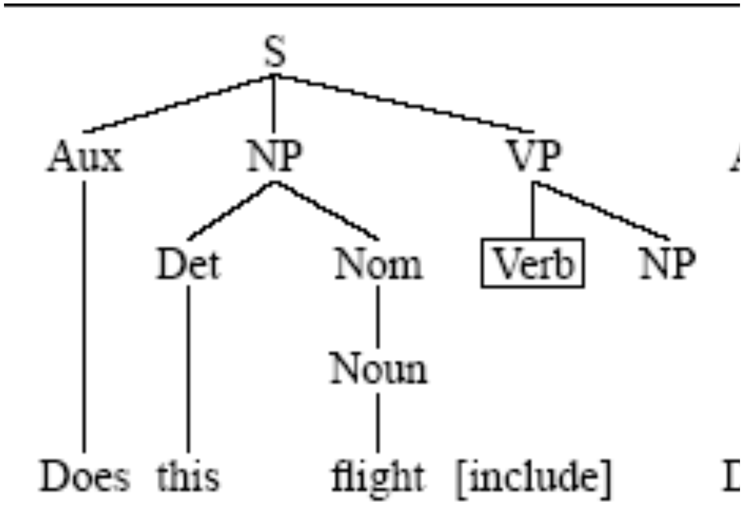


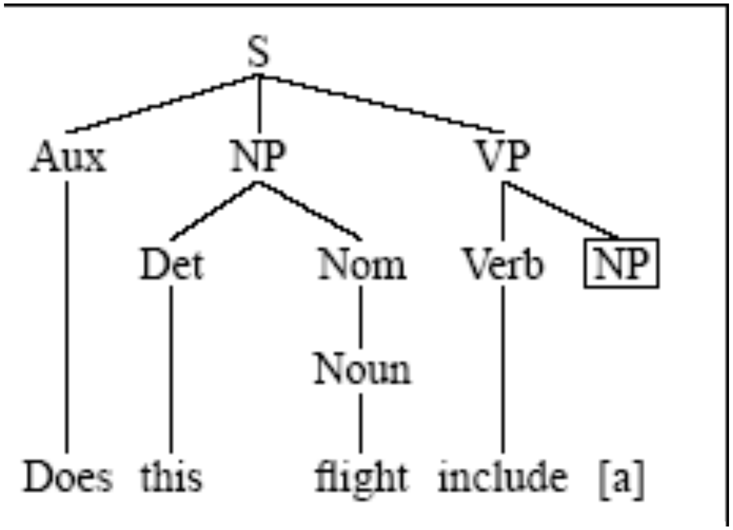


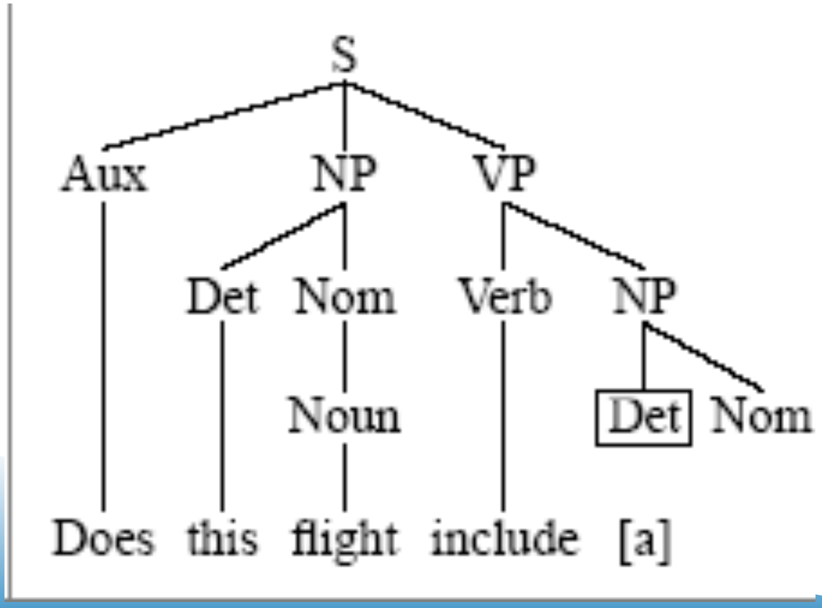


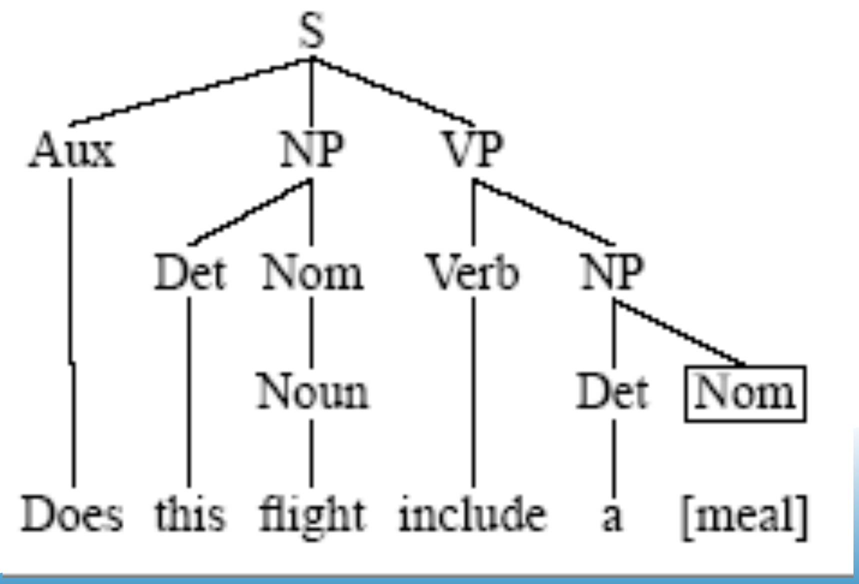


TOP-DOWN, DEPTH-FIRST, LEFT-TO-RIGHT (III)

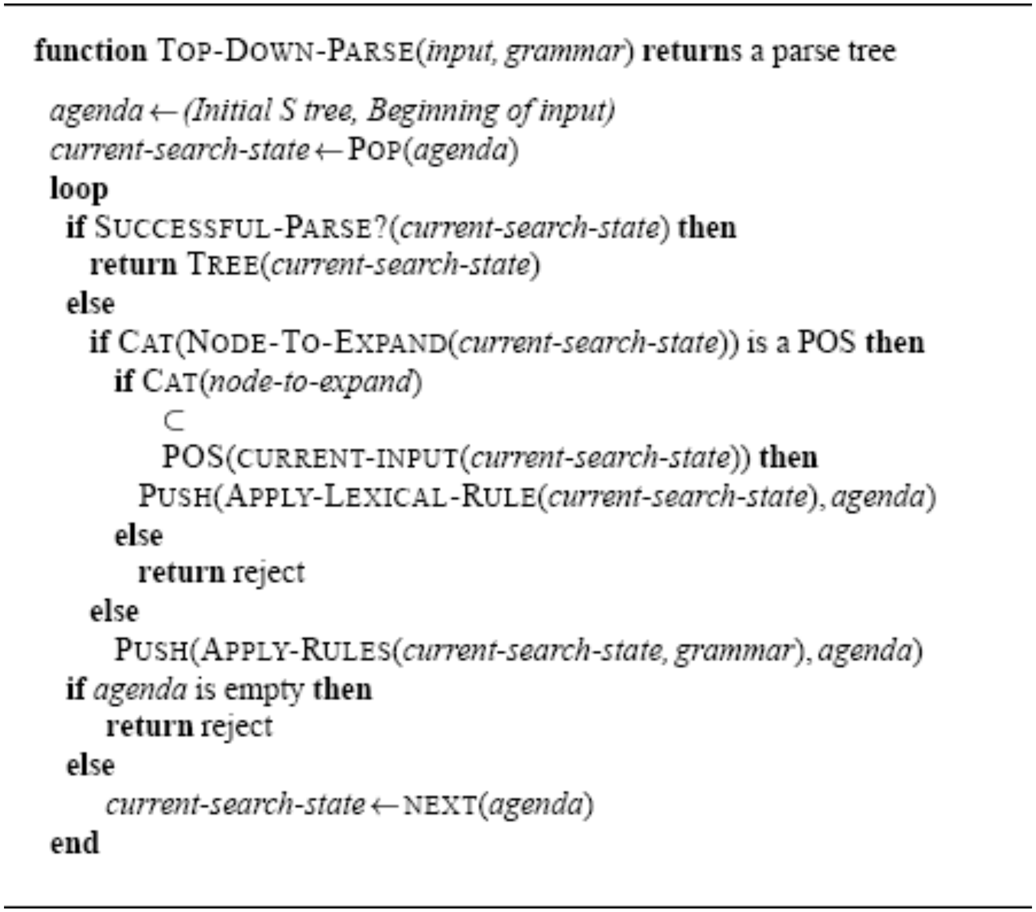








A T-D, D-F, L-R PARSER



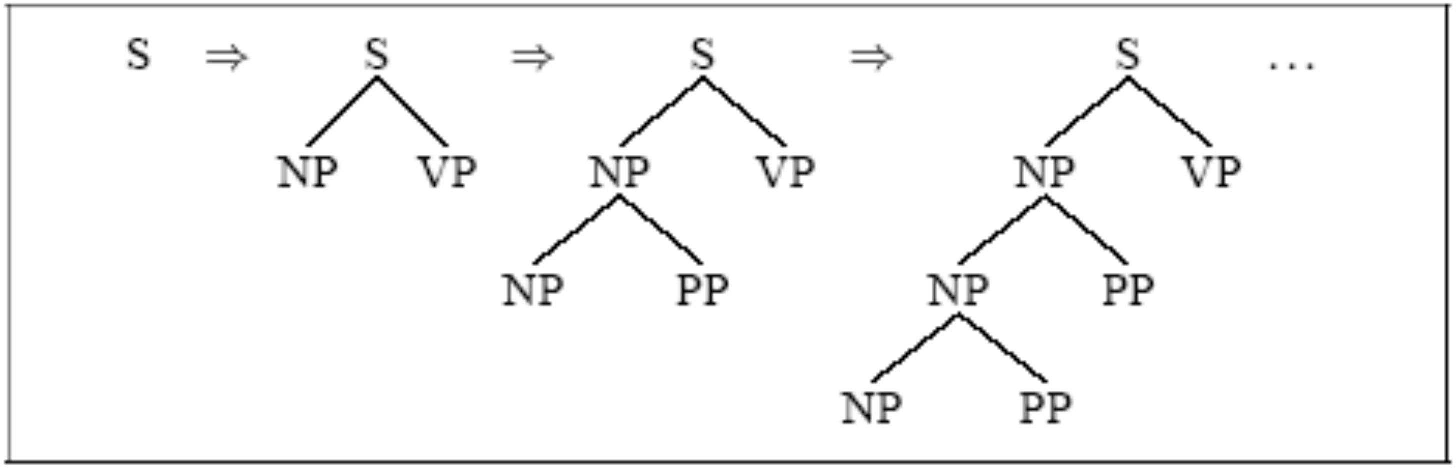
```
function TOP-DOWN-PARSE(input, grammar) returns a parse tree

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

LEFT-RECURSION

- A LEFT-RECURSIVE grammar may cause a T-D, D-F, L-R parser to never return
- Examples of left-recursive rules:
  - NP → NP PP
  - S → S and S
  - But also:
    - NP → Det Nom
    - Det → NP’s

THE PROBLEM WITH LEFT-RECURSION

NP → NP PP



Dynamic Programming

- We need a method that fills a table with partial results that
  - Does not do (avoidable) repeated work
  - Does not fall prey to left-recursion
  - Can find all the pieces of an exponential number of trees in polynomial time.
- Two popular methods
  - CKY
  - Earley

The CKY (Cocke-Kasami-Younger) Algorithm

- Requires the grammar be in Chomsky Normal Form (CNF)
  - All rules must be in following form:
    - A -> B C
    - A -> w
- Any grammar can be converted automatically to Chomsky Normal Form

Converting to CNF

- Rules that mix terminals and non-terminals
  - Introduce a new dummy non-terminal that covers the terminal
    - INFVP -> to VP replaced by:

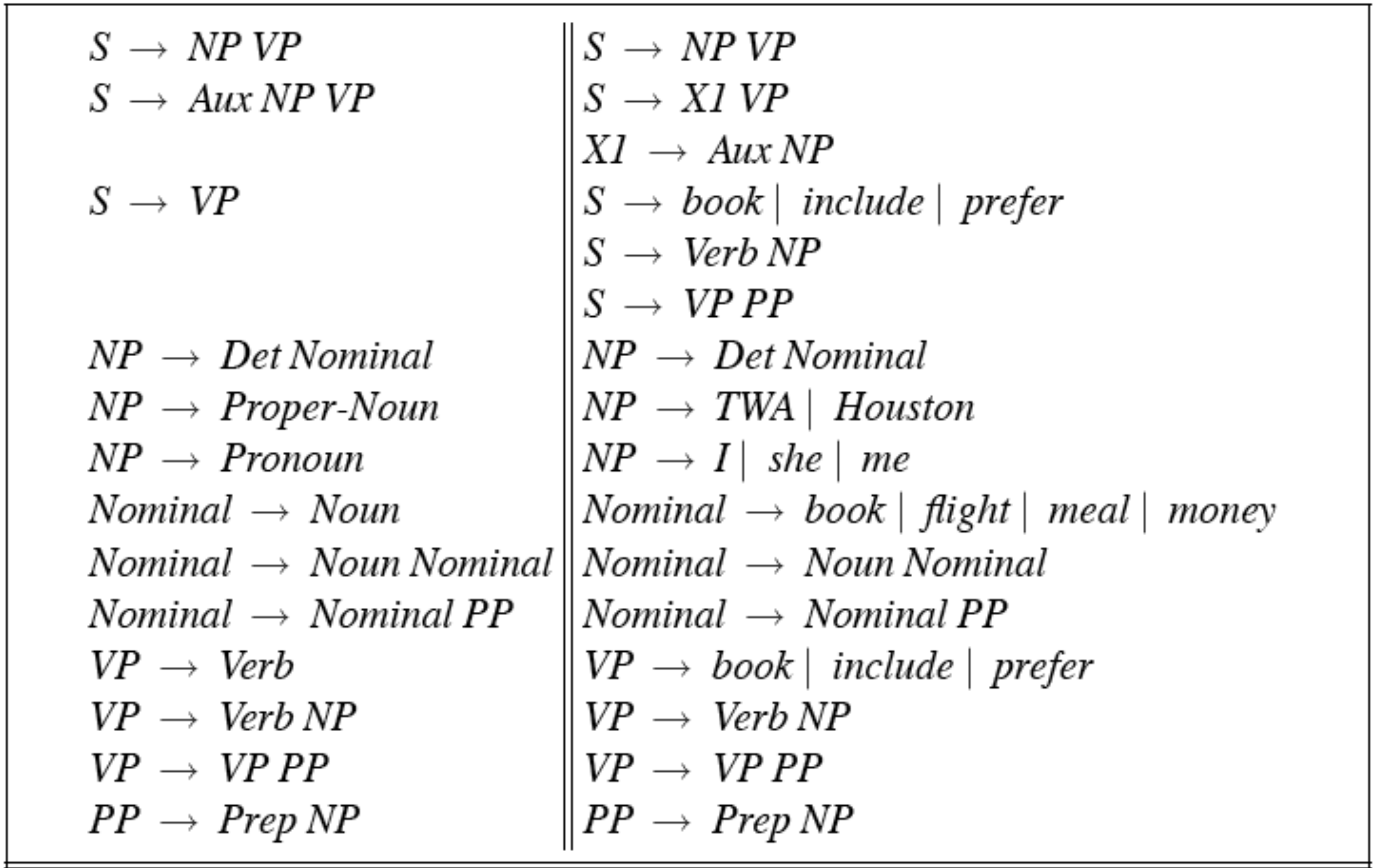
    - INFVP -> TO VP
    - TO -> to
- Rules that have a single non-terminal on right ("unit productions")
  - Rewrite each unit production with the RHS of their expansions
- Rules whose right hand side length >2
  - Introduce dummy non-terminals that spread the right-hand side

Sample Grammar

S → NP VP
S→ Aux NP VP
S → VP
NP → NP PP
NP → Det Noun
NP → PrN
VP → V
VP → V NP
VP → V NP PP
PP → Prep NP

Det → | a | the
Noun→ book | saw | mark
Verb → book | saw
Proper-Noun → Mark
Aux→ Did | Has
Prep→ to | on | near

Automatic Conversion to CNF



S → NP VP	S → NP VP
S → Aux NP VP	S → X1 VP
	X1 → Aux NP
S → VP	S → book | include | prefer
	S → Verb NP
	S → VP PP
NP → Det Nominal	NP → Det Nominal
NP → Proper-Noun	NP → TWA | Houston
NP → Pronoun	NP → I | she | me
Nominal → Noun	Nominal → book | flight | meal | money
Nominal → Noun Nominal	Nominal → Noun Nominal
Nominal → Nominal PP	Nominal → Nominal PP
VP → Verb	VP → book | include | prefer
VP → Verb NP	VP → Verb NP
VP → VP PP	VP → VP PP
PP → Prep NP	PP → Prep NP

Figure 10.15 Original L0 Grammar and its conversion to CNF

Back to CKY Parsing

- Given rules in CNF
- Consider the rule A -> BC
  - If there is an A in the input then there must be a B followed by a C in the input.
  - If the A goes from i to j in the input then there must be some k st. i<k<j
    - Ie. The B splits from the C someplace.

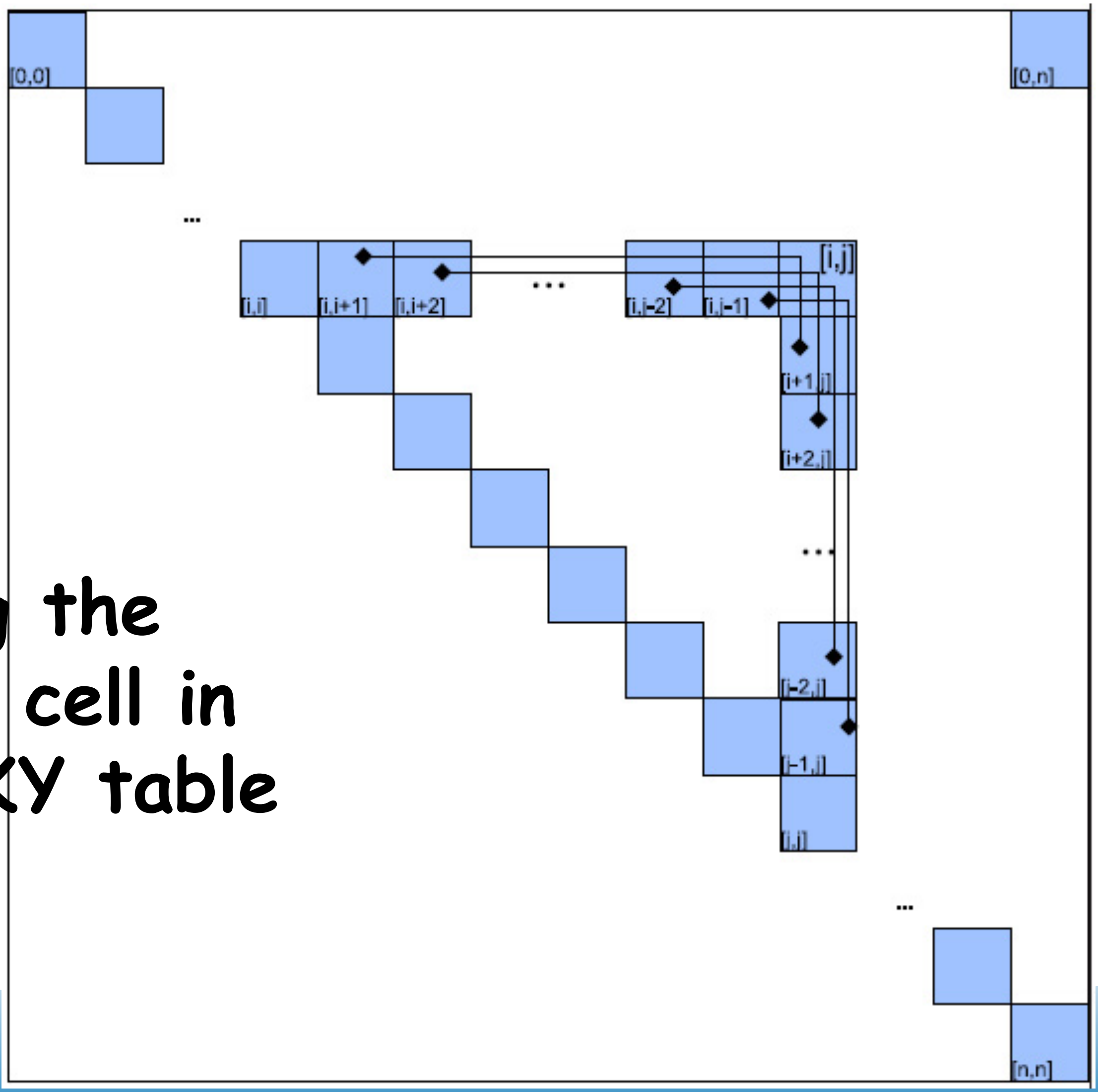
CKY

- So let's build a table so that an A spanning from i to j in the input is placed in cell [i,j] in the table.
- So a non-terminal spanning an entire string will sit in cell [0, n]
- If we build the table bottom up we'll know that the parts of the A must go from i to k and from k to j

CKY

- Meaning that for a rule like A -> B C we should look for a B in [i,k] and a C in [k,j].
- In other words, if we think there might be an A spanning i,j in the input… AND
- A -> B C is a rule in the grammar THEN
- There must be a B in [i,k] and a C in [k,j] for some i<k<j
- So just loop over the possible k values

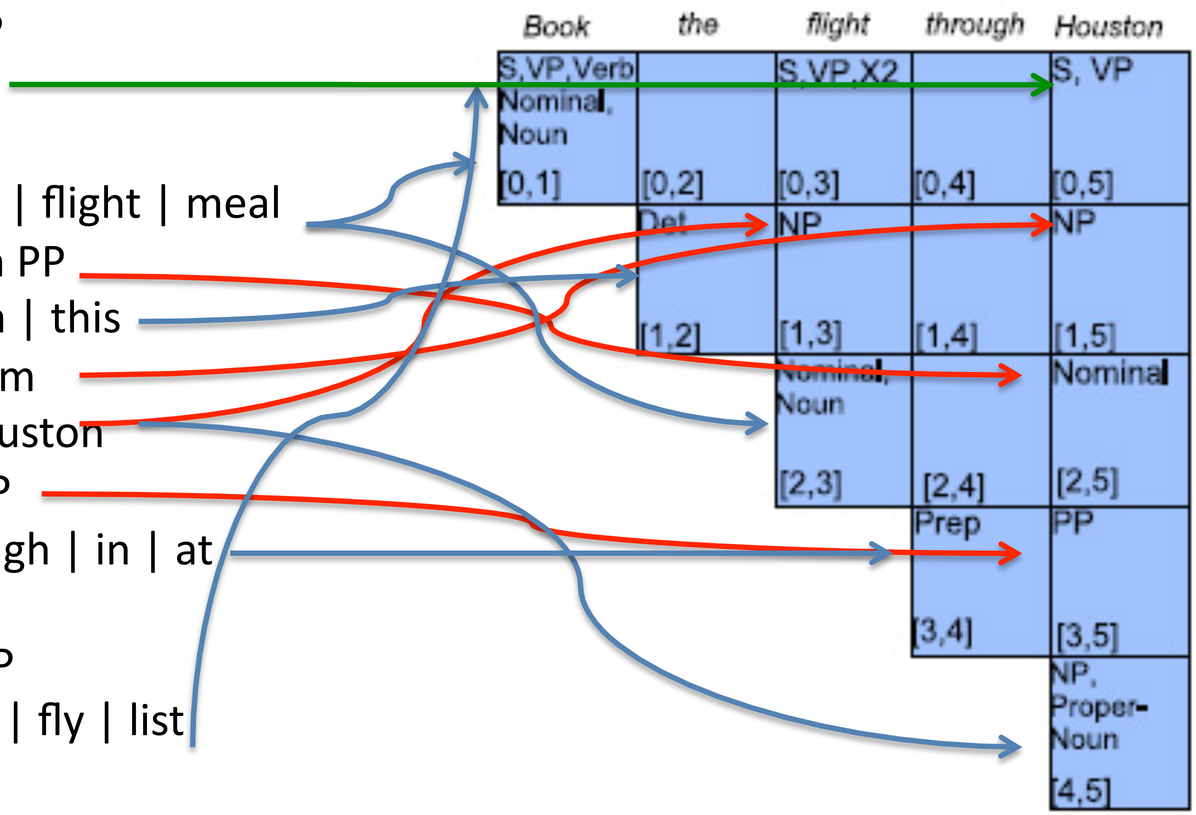
CKY Table



- Filling the [i,j]th cell in the CKY table

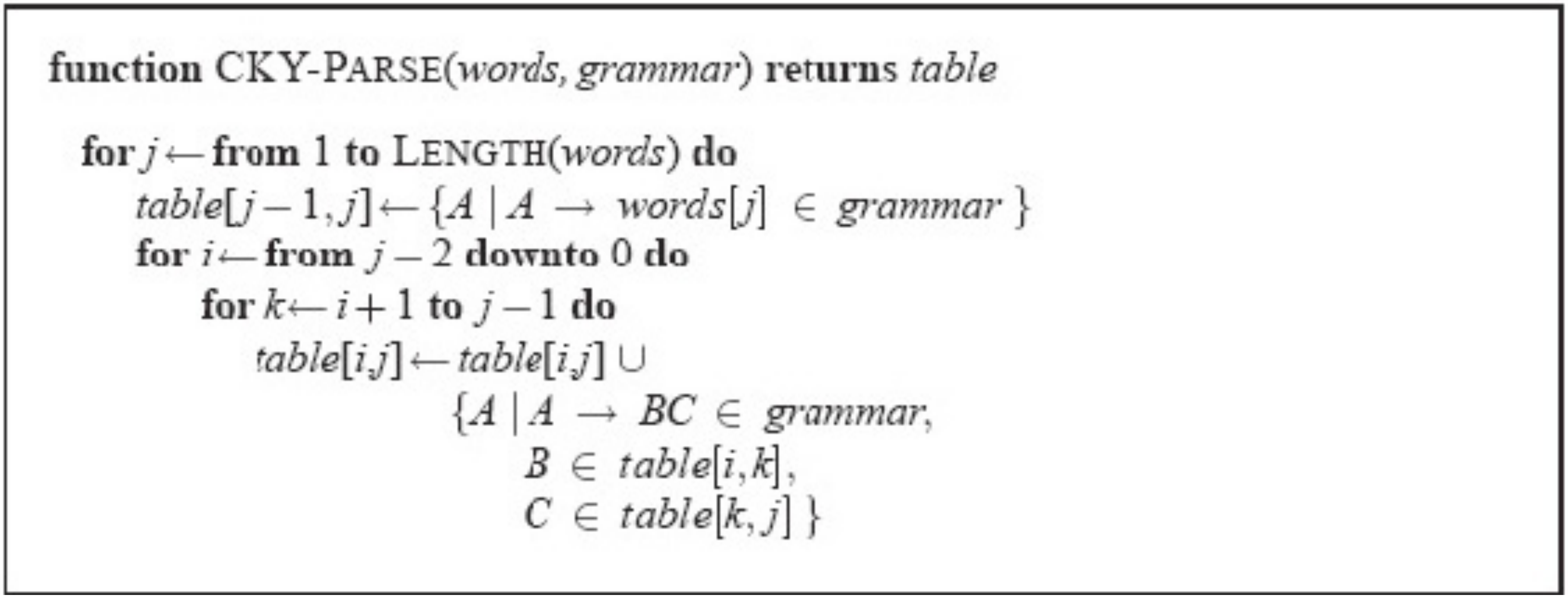
0 Book 1 the 2 flight 3 through 4 Houston 5

S → NP VP
S → X1 VP
X1 → AUX NP
S → Verb NP
S → VP PP
Nom → book | flight | meal
Nom → Nom PP
Det → the | a | this
NP → Det Nom
NP → twa houston
PP → Prep NP
Prep → through | in | at
VP → Verb
VP → Verb NP
Verb → book | fly | list



Book	*the*	*flight*	*through*	*Houston*
S,VP,Verb Nominal, Noun [0,1]	[0,2]	S,VP,X2 [0,3]	[0,4]	S, VP [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 Algorithm

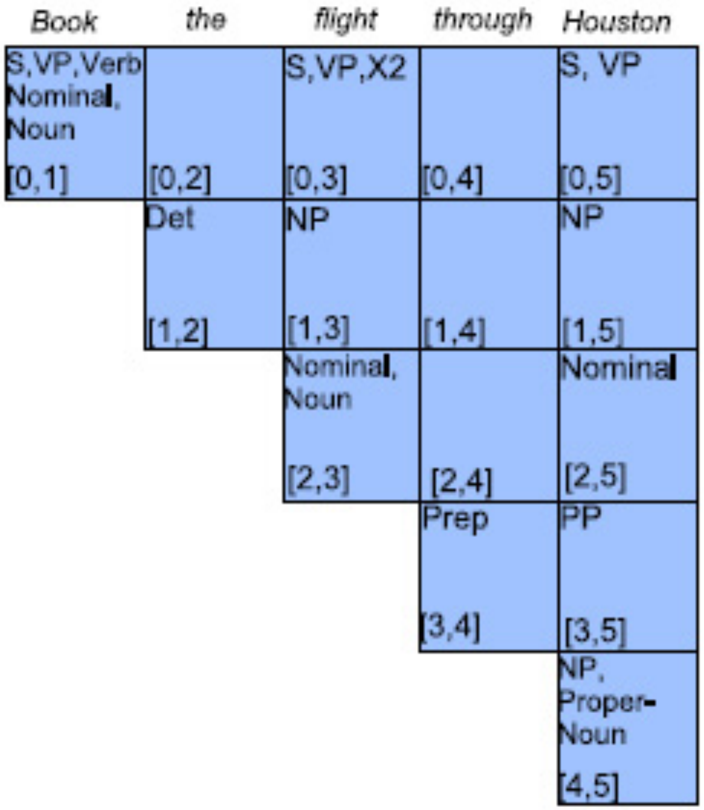


```
function CKY-PARSE(words, grammar) returns table

  for $j \leftarrow$ from 1 to LENGTH(words) do
      $table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$
      for $i \leftarrow$ from $j-2$ downto 0 do
          for $k \leftarrow i+1$ to $j-1$ do
              $table[i,j] \leftarrow table[i,j] \cup$
                        $\{A \mid A \rightarrow BC \in grammar,$
                             $B \in table[i,k],$
                             $C \in table[k,j]\}$
```

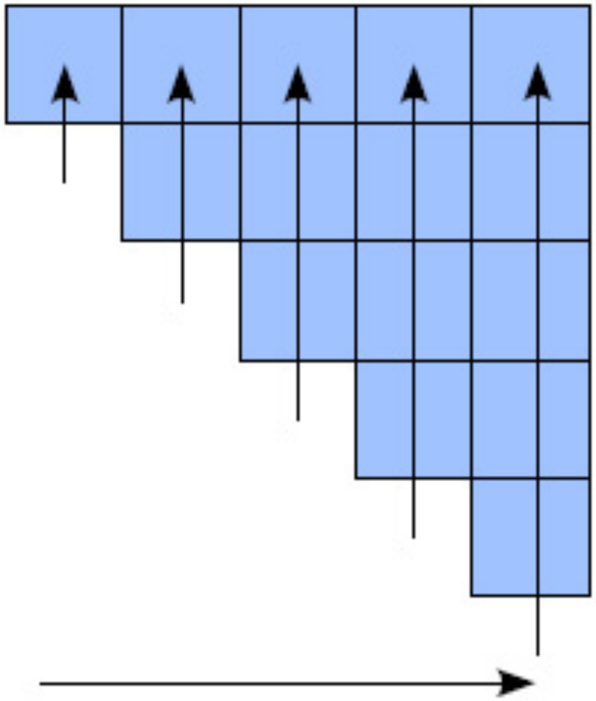
Note

- We arranged the loops to fill the table a column at a time, from left to right, bottom to top.
  - This assures us that whenever we're filling a cell, the parts needed to fill it are already in the table (to the left and below)
  - Are there other ways to fill the table?

0 Book 1 the 2 flight 3 through 4 Houston 5

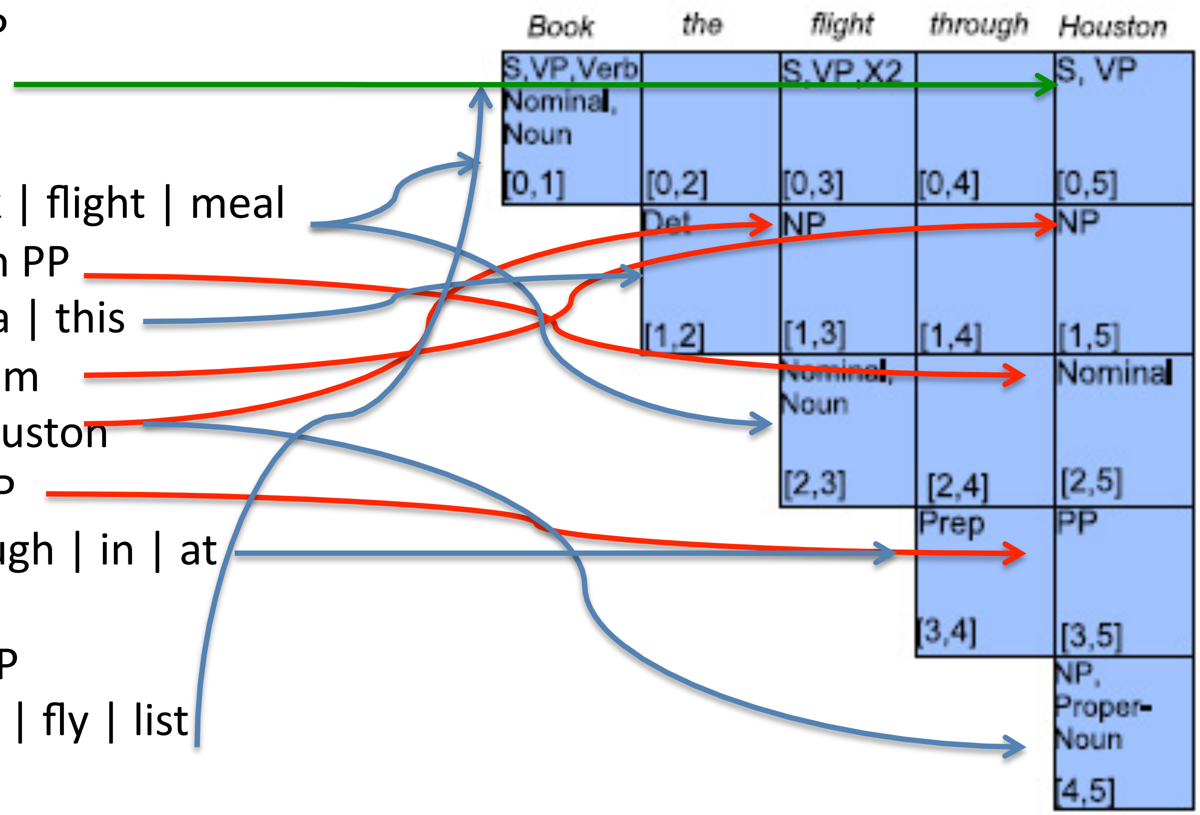


Book	the	flight	through	Houston
S,VP,Verb Nominal, Noun [0,1]	[0,2]	S,VP,X2 [0,3]	[0,4]	S, VP [0,5]
	Det [1,2]	NP [1,3]	[1,4]	NP [1,5]
		Nominal, Noun [2,3]	[2,4]	Nominal [2,5]
			Prep [3,4]	PP [3,5]
				NP, Proper- Noun [4,5]



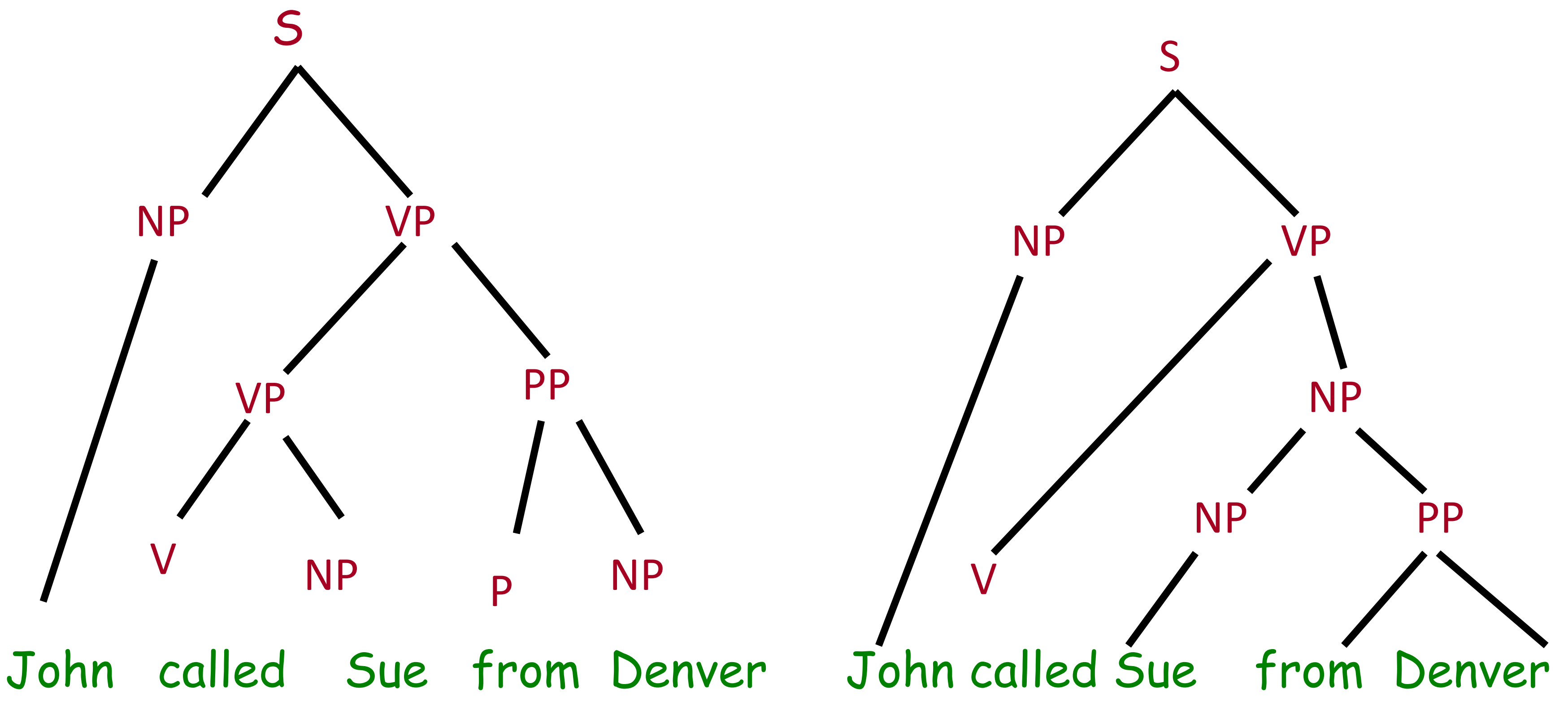
0 Book 1 the 2 flight 3 through 4 Houston 5

S → NP VP
S → X1 VP
X1 → AUX NP
S → Verb NP
S → VP PP
Nom → book | flight | meal
Nom → Nom PP
Det → the | a | this
NP → Det Nom
NP → twa houston
PP → Prep NP
Prep → through | in | at
VP → Verb
VP → Verb NP
Verb → book | fly | list



Book	the	flight	through	Houston
S,VP,Verb Nominal, Noun [0,1]	[0,2]	S,VP,X2 [0,3]	[0,4]	S, VP [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]

Example


S
NP
VP
VP
PP
V
NP
P
NP
John called Sue from Denver
S
NP
VP
NP
V
NP
PP
John called Sue from Denver

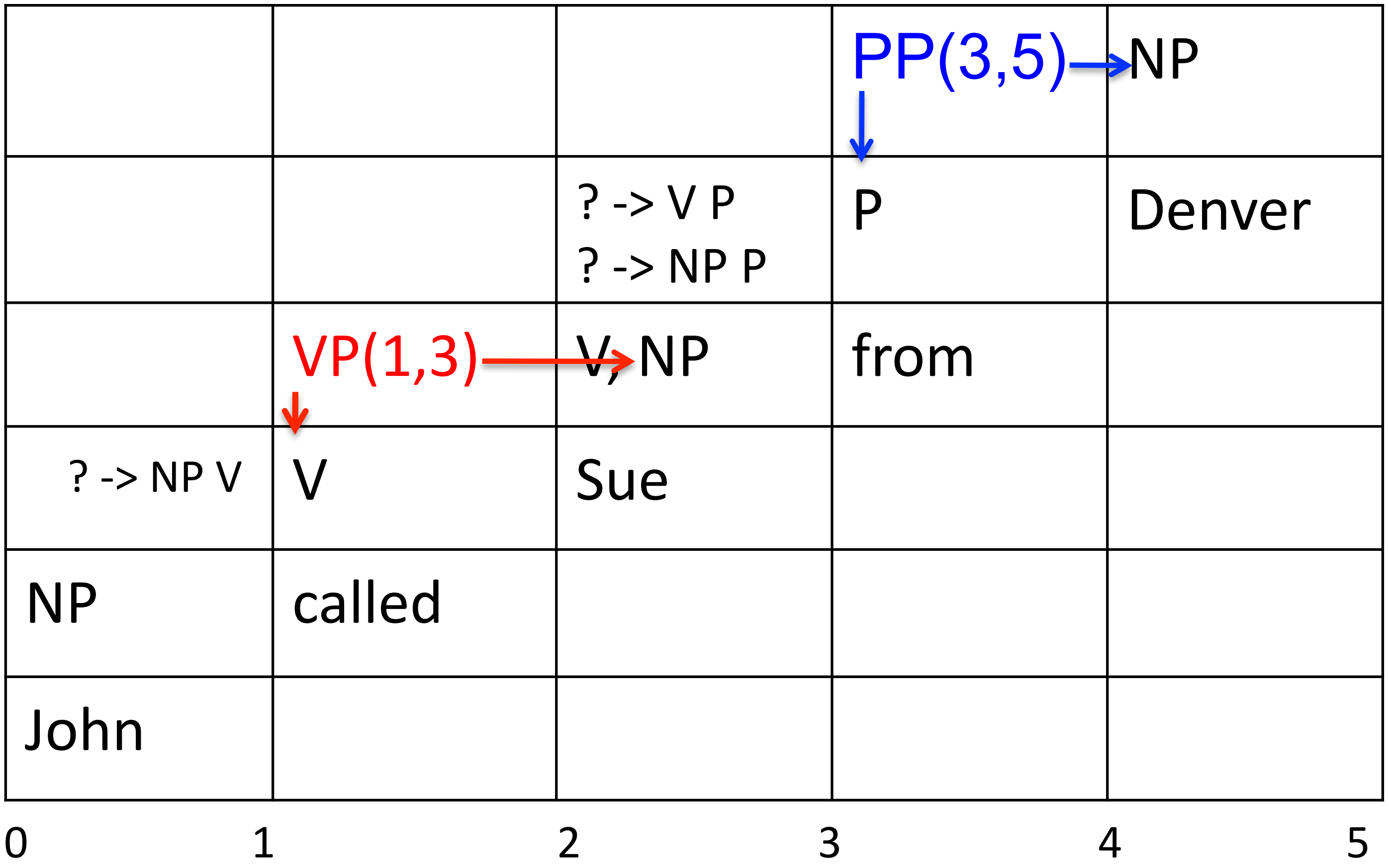
Example 1

S -> NP VP
VP -> V NP
VP -> VP PP
NP -> NP PP
PP -> P NP
NP -> John
NP -> Sue
NP -> Denver
V -> called
V -> sue
P -> from

S(0,5)				NP(4,5)
			P(3,4)	Denver
		NP(2,3) V(2,3)	from	
	V(1,2)	Sue		
NP(0,1)	called			
John				

Example 2

S -> NP VP
NP -> NP PP
VP -> V NP★
VP -> VP PP
PP -> P NP★
NP -> John
NP -> Sue
NP -> Denver
V -> called
V -> sue
P -> from

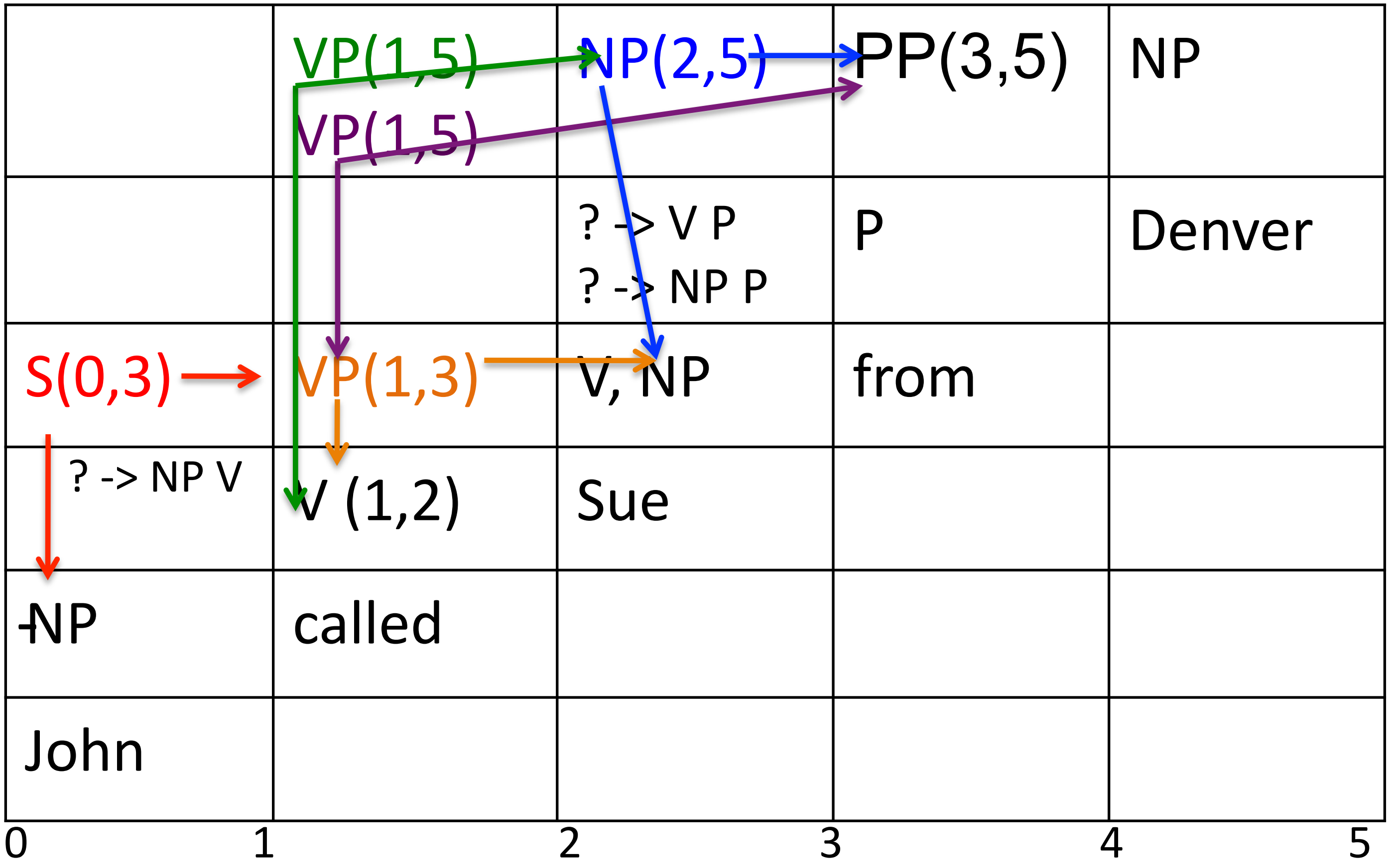


			PP(3,5) →	NP
		? -> V P ? -> NP P	P	Denver
	VP(1,3) →	V, NP	from	
? -> NP V	V	Sue		
NP	called			
John				

0 1 2 3 4 5

Example 3

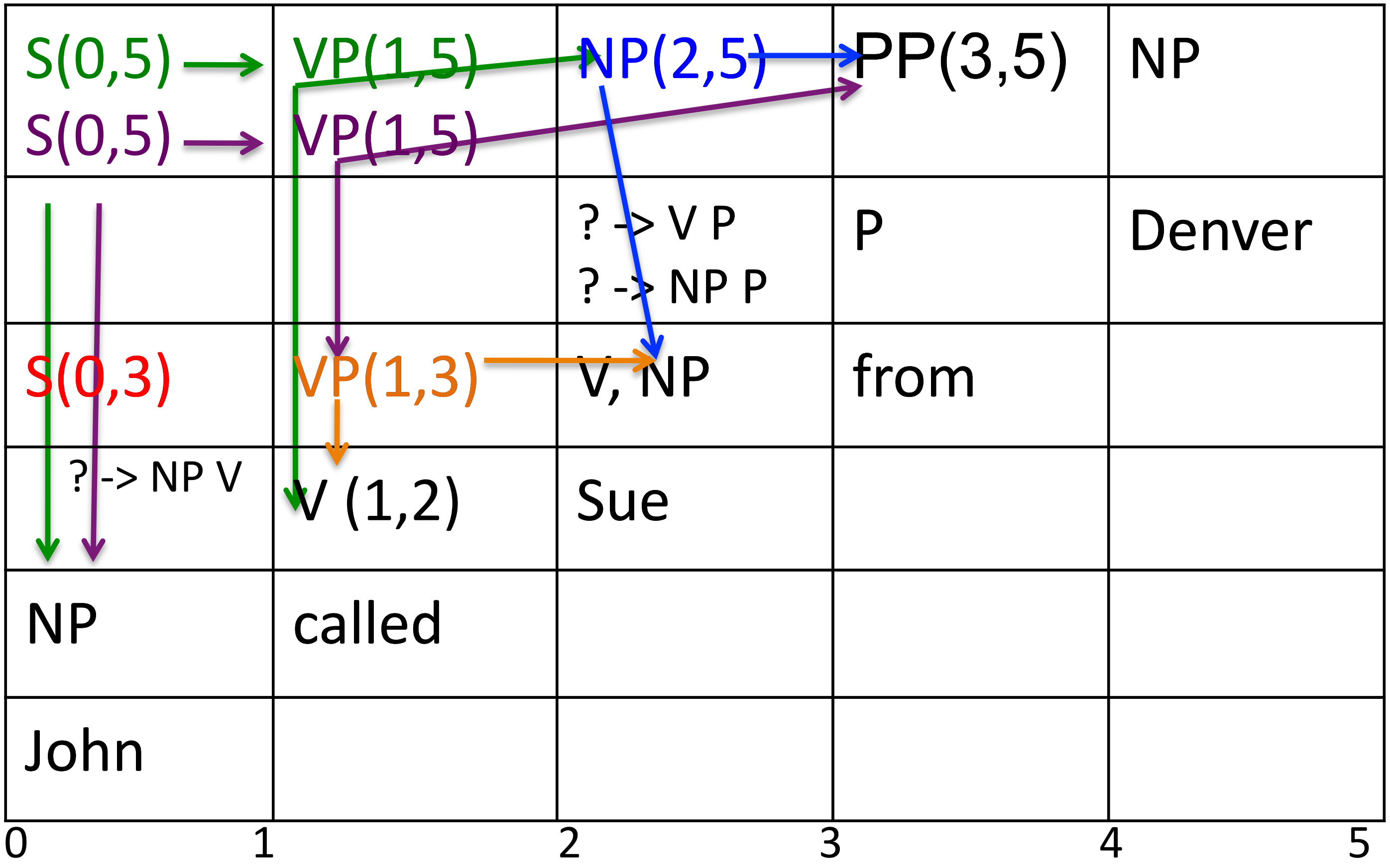
S -> NP VP★
VP -> V NP★★
NP -> NP PP★
VP -> VP PP★
PP -> P NP
NP -> John
NP -> Mary NP -> Denver
V -> called
P -> from



	VP(1,5) VP(1,5)	NP(2,5)	PP(3,5)	NP
		? -> V P ? -> NP P	P	Denver
S(0,3)	VP(1,3)	V, NP	from	
? -> NP V	V (1,2)	Sue		
NP	called			
John				

Example 4

S -> NP VP★★
VP -> V NP
NP -> NP PP
VP -> VP PP
PP -> P NP
NP -> John
NP -> Sue
NP -> Denver
V -> called
V -> sue
P -> from



S(0,5) → S(0,5) →	VP(1,5) VP(1,5)	NP(2,5)	PP(3,5)	NP
		? -> V P ? -> NP P	P	Denver
S(0,3)	VP(1,3)	V, NP	from	
? -> NP V	V (1,2)	Sue		
NP	called			
John				

Back to Ambiguity

- Did we solve it?
- No…
  - Both CKY and Earley will result in multiple S structures for the [0,n] table entry.
  - They both efficiently store the sub-parts that are shared between multiple parses.
  - But neither can tell us which one is right.
  - Not a parser – a recognizer
    - The presence of an S state with the right attributes in the right place indicates a successful recognition.
    - But no parse tree… no parser
    - That's how we solve (not) an exponential problem in polynomial time

Converting CKY from Recognizer to Parser

- With the addition of a few pointers we have a parser
- Augment each new cell in chart to point to where we came from.

Problem (minor)

- We said CKY requires the grammar to be binary (ie. In Chomsky-Normal Form)
- We showed that any arbitrary CFG can be converted to Chomsky-Normal Form so that’s not a huge deal
- Except when you change the grammar the trees come out wrong
- All things being equal we’d prefer to leave the grammar alone.

Earley Parsing

- Allows arbitrary CFGs
- Where CKY is bottom-up, Earley is top-down
- Fills a table in a single sweep over the input words
  - Table is length N+1; N is number of words
  - Table entries represent
    - Completed constituents and their locations
    - In-progress constituents
    - Predicted constituents

States

- The table-entries are called states and are represented with dotted-rules.

S -> · VP	A VP is predicted
NP -> Det · Nominal	An NP is in progress
VP -> V NP ·	A VP has been found

States/Locations

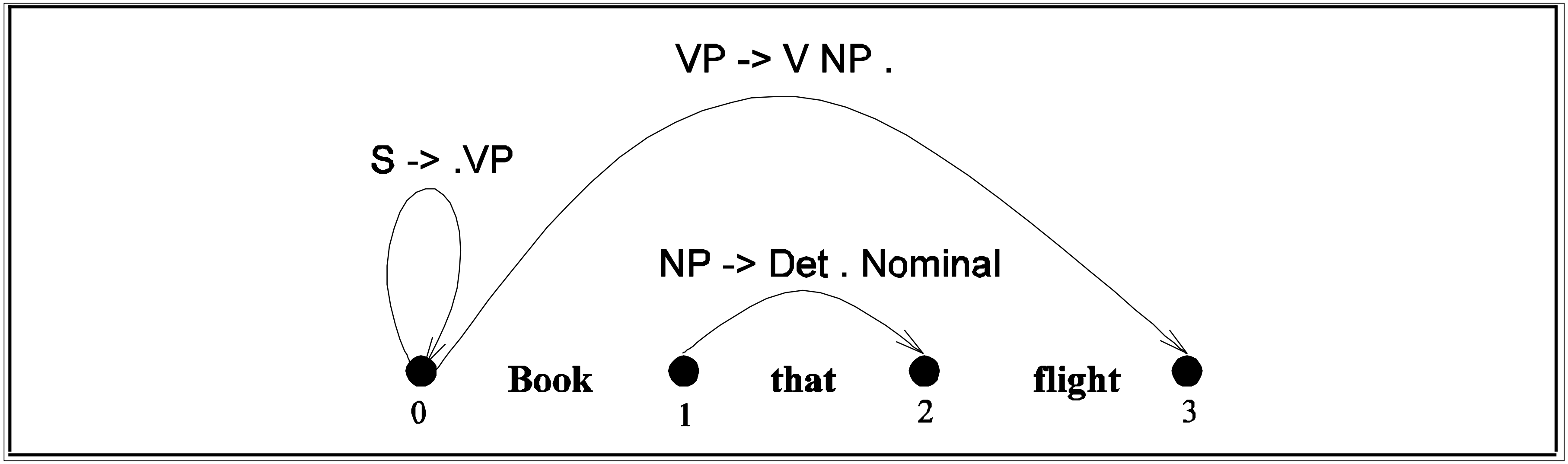
- It would be nice to know where these things are in the input so…

  S -> · VP [0,0] A VP is predicted at the start of the sentence

  NP -> Det · Nominal [1,2] An NP is in progress; the Det goes from 1 to 2

  VP -> V NP · [0,3] A VP has been found starting at 0 and ending at 3

Graphically



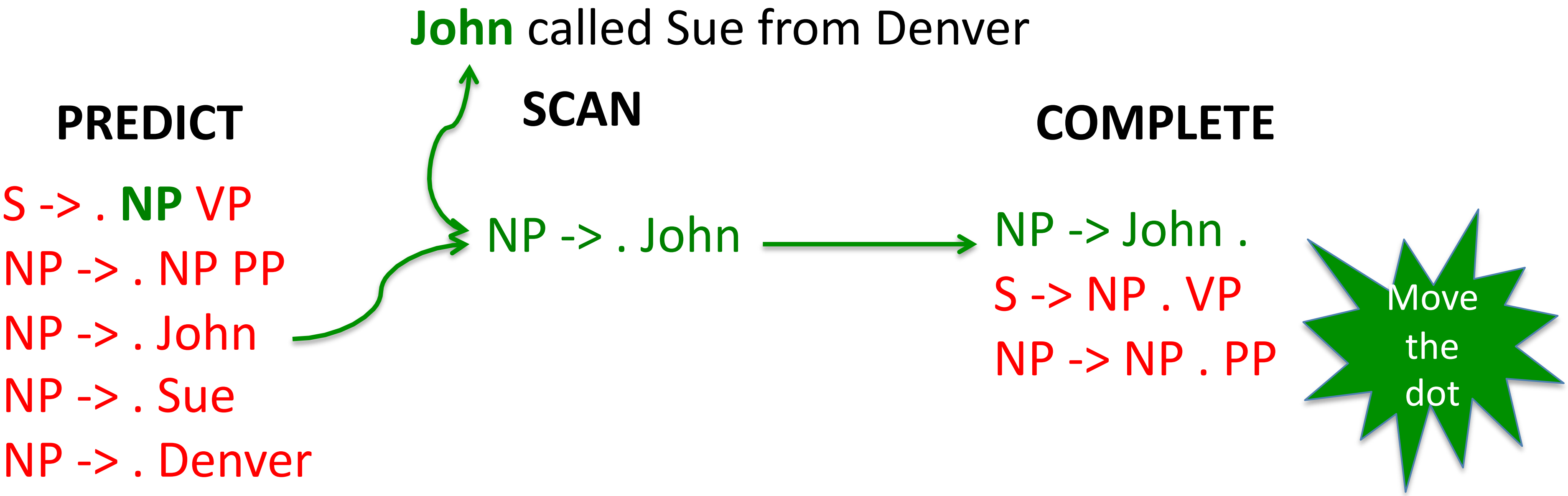
Earley

- As with most dynamic programming approaches, the answer is found by looking in the table in the right place.
- In this case, there should be an S state in the final column that spans from 0 to n+1 and is complete.
- If that’s the case you’re done.
  - S –> α · [0,n+1]

Earley Algorithm

- March through chart left-to-right.
- At each step, apply 1 of 3 operators
  - Predictor
    - Create new states representing top-down expectations
  - Scanner
    - Match word predictions (rule with word after dot) to words
  - Completer
    - When a state is complete, see what rules were looking for that completed constituent

Earley's example 1
Predict - Scan- Complete



John called Sue from Denver

PREDICT

S -> . **NP** VP
NP -> . NP PP
NP -> . John
NP -> . Sue
NP -> . Denver

SCAN

NP -> . John

COMPLETE

NP -> John .
S -> NP . VP
NP -> NP . PP

Move the dot

Rules not predicted

P -> . V NP
VP -> . VP PP
PP -> . P NP
V -> . called
V ->. sue
P -> . from

NOTE TO SELF:
Put in spans

Earley's example 2

John **called** Sue from Denver

PREDICT	**SCAN**	**COMPLETE**
S -> NP . VP	**V -> . called**	**V -> called .**
NP -> NP . PP		**VP -> V . NP**
VP -> . V NP		
VP -> . VP PP		
PP -> . P NP		
V -> . called		
V ->. sue		
P -> . from		

Earley's example 3

John called **Sue** from Denver

PREDICT

S -> NP . VP
NP -> NP . PP
VP -> V . NP
VP -> . VP PP
PP -> . P NP
NP -> . John
NP -> . Sue
NP -> . Denver

SCAN

NP -> . Sue

COMPLETE

NP -> Sue .
VP -> V NP .
VP -> VP . PP
S -> NP VP .

Am I done?

Earley’s example 4

John called Sue **from Denver**

S -> NP . VP
NP -> NP . PP
VP -> V . NP
VP -> VP . PP
PP -> . P NP
P -> . from

NP -> . John
NP -> . Sue
NP -> . Denver

P -> . from

NP -> . Denver

S -> NP . VP
NP -> NP . PP
VP -> VP . PP
PP -> P . NP
P -> from .

DONE

NP -> Denver .
PP -> P NP .
NP -> NP PP .
VP -> VP PP .
VP -> V NP .
S -> NP VP .

Predictor

- Given a state
  - With a non-terminal to right of dot
  - That is not a part-of-speech category
  - Create a new state for each expansion of the non-terminal
  - Place these new states into same chart entry as generated state, beginning and ending where generating state ends.
  - So predictor looking at
    - S -> . VP [0,0]
  - results in
    - VP -> . Verb [0,0]
    - VP -> . Verb NP [0,0]

Scanner

- Given a state
  - With a non-terminal to right of dot
  - That is a part-of-speech category
  - If the next word in the input matches this part-of-speech
  - Create a new state with dot moved over the non-terminal
  - So scanner looking at
    - VP -> . Verb NP [0,0]
  - If the next word, “book”, can be a verb, add new state:
    - VP -> Verb . NP [0,1]
  - Add this state to chart entry following current one
  - Note: Earley algorithm uses top-down input to disambiguate POS! Only POS predicted by some state can get added to chart!

Completer

- Applied to a state when its dot has reached right end of role.
- Parser has discovered a category over some span of input.
- Find and advance all previous states that were looking for this category
  - copy state, move dot, insert in current chart entry
- Given:
  - NP -> Det Nominal . [1,3]
  - VP -> Verb. NP [0,1]
- Add
  - VP -> Verb NP . [0,3]

Earley: how do we know we are done?

- How do we know when we are done?
- Find an S state in the final column that spans from 0 to n+1 and is complete.
- If that's the case you're done.
  - $S \rightarrow \alpha \cdot [0,n+1]$

Earley

- So sweep through the table from 0 to n+1…
  - New predicted states are created by starting top-down from S
  - New incomplete states are created by advancing existing states as new constituents are discovered
  - New complete states are created in the same way.

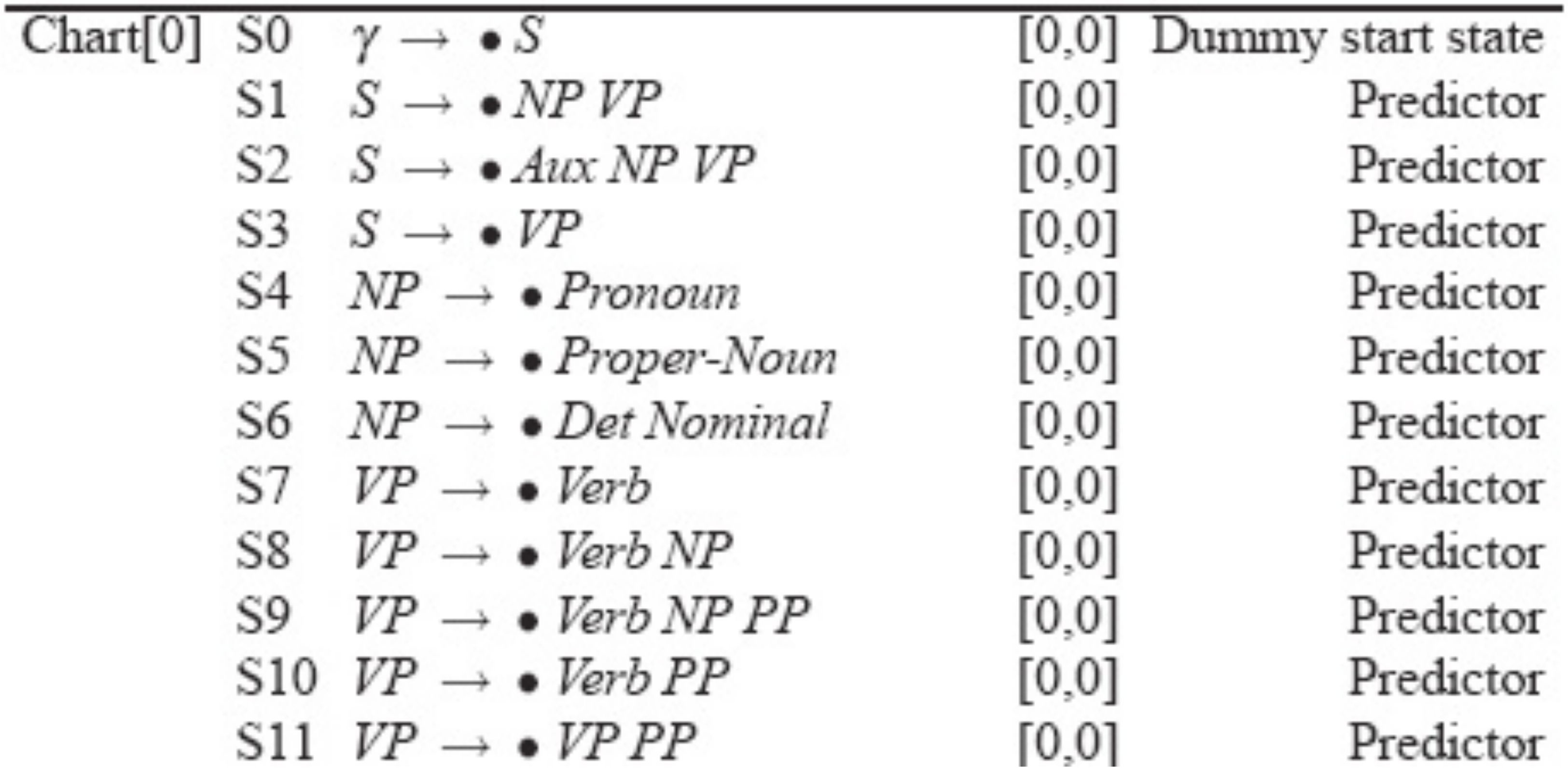
Earley

- More specifically…
  1. Predict all the states you can upfront
  2. Read a word
     1. Extend states based on matches
     2. Add new predictions
     3. Go to 2
  3. Look at N+1 to see if you have a winner

Example

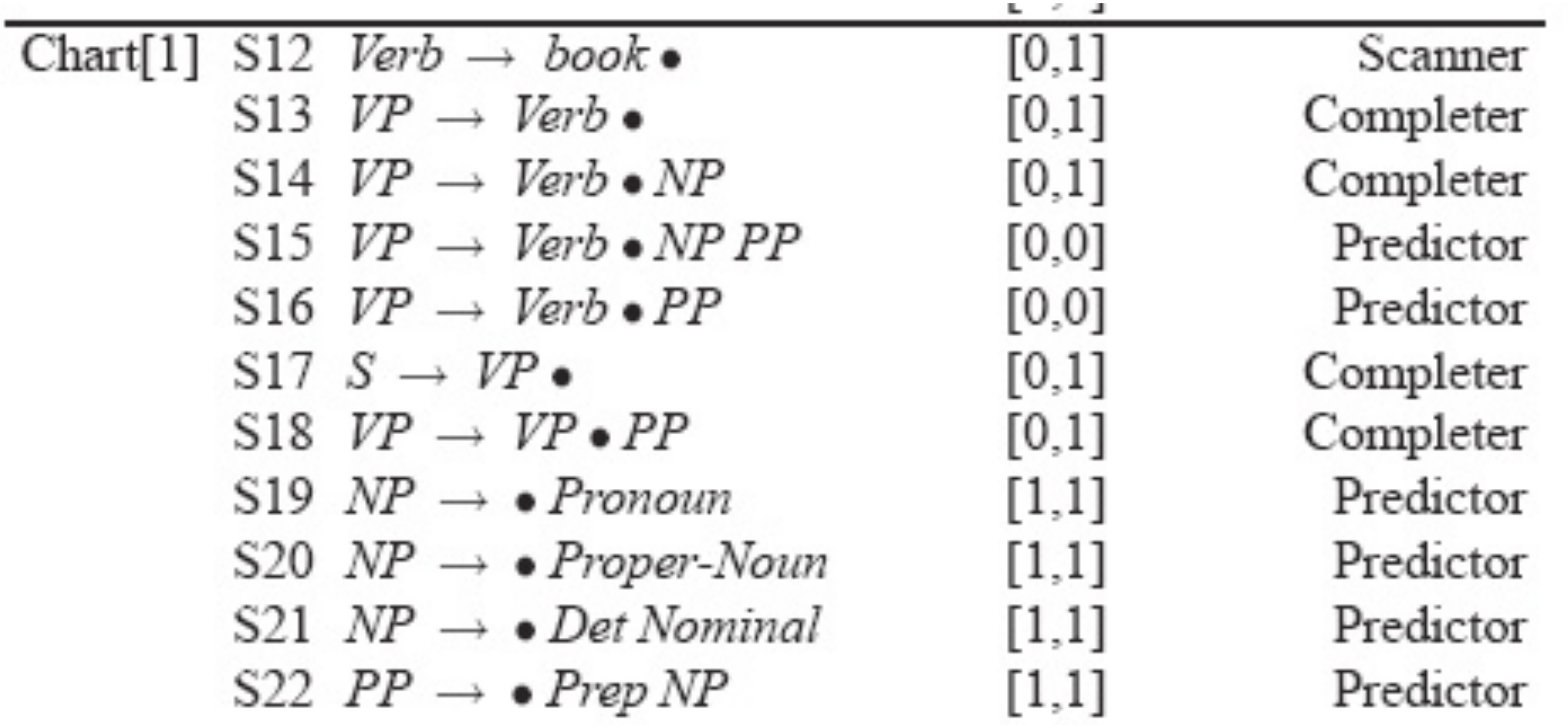
- Book that flight
- We should find… an S from 0 to 3 that is a completed state…

Example



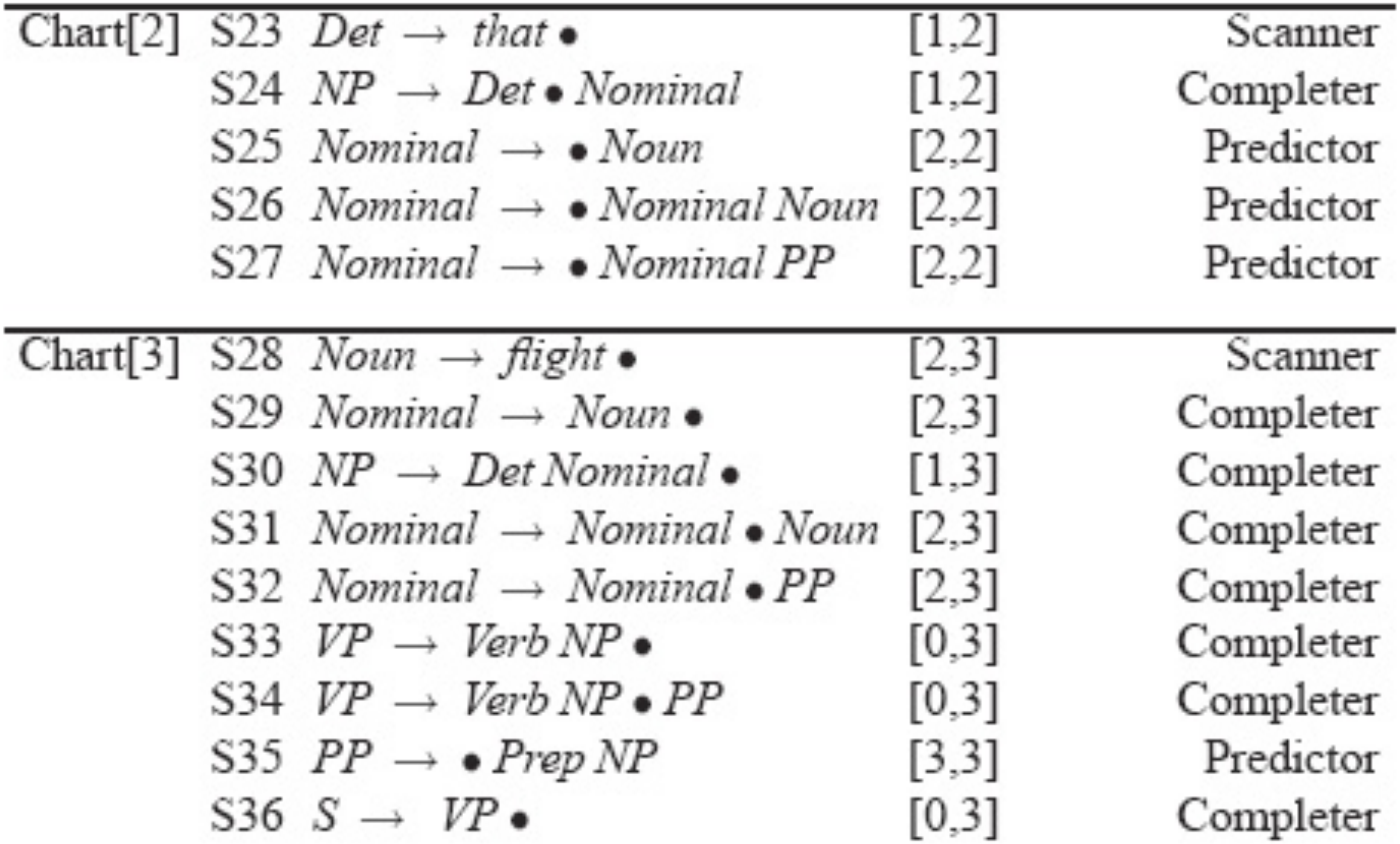
Chart[0]	S0	$\gamma \rightarrow \bullet S$	[0,0]	Dummy start state
	S1	$S \rightarrow \bullet NP\ VP$	[0,0]	Predictor
	S2	$S \rightarrow \bullet Aux\ NP\ VP$	[0,0]	Predictor
	S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
	S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
	S5	$NP \rightarrow \bullet Proper\text{-}Noun$	[0,0]	Predictor
	S6	$NP \rightarrow \bullet Det\ Nominal$	[0,0]	Predictor
	S7	$VP \rightarrow \bullet Verb$	[0,0]	Predictor
	S8	$VP \rightarrow \bullet Verb\ NP$	[0,0]	Predictor
	S9	$VP \rightarrow \bullet Verb\ NP\ PP$	[0,0]	Predictor
	S10	$VP \rightarrow \bullet Verb\ PP$	[0,0]	Predictor
	S11	$VP \rightarrow \bullet VP\ PP$	[0,0]	Predictor

Example



Chart[1]	S12	$Verb \rightarrow book \bullet$	[0,1]	Scanner
	S13	$VP \rightarrow Verb \bullet$	[0,1]	Completer
	S14	$VP \rightarrow Verb \bullet NP$	[0,1]	Completer
	S15	$VP \rightarrow Verb \bullet NP\ PP$	[0,0]	Predictor
	S16	$VP \rightarrow Verb \bullet PP$	[0,0]	Predictor
	S17	$S \rightarrow VP \bullet$	[0,1]	Completer
	S18	$VP \rightarrow VP \bullet PP$	[0,1]	Completer
	S19	$NP \rightarrow \bullet Pronoun$	[1,1]	Predictor
	S20	$NP \rightarrow \bullet Proper\text{-}Noun$	[1,1]	Predictor
	S21	$NP \rightarrow \bullet Det\ Nominal$	[1,1]	Predictor
	S22	$PP \rightarrow \bullet Prep\ NP$	[1,1]	Predictor

Example



Chart[2]	S23	$Det \rightarrow that \bullet$	[1,2]	Scanner
	S24	$NP \rightarrow Det \bullet Nominal$	[1,2]	Completer
	S25	$Nominal \rightarrow \bullet Noun$	[2,2]	Predictor
	S26	$Nominal \rightarrow \bullet Nominal\ Noun$	[2,2]	Predictor
	S27	$Nominal \rightarrow \bullet Nominal\ PP$	[2,2]	Predictor
Chart[3]	S28	$Noun \rightarrow flight \bullet$	[2,3]	Scanner
	S29	$Nominal \rightarrow Noun \bullet$	[2,3]	Completer
	S30	$NP \rightarrow Det\ Nominal \bullet$	[1,3]	Completer
	S31	$Nominal \rightarrow Nominal \bullet Noun$	[2,3]	Completer
	S32	$Nominal \rightarrow Nominal \bullet PP$	[2,3]	Completer
	S33	$VP \rightarrow Verb\ NP \bullet$	[0,3]	Completer
	S34	$VP \rightarrow Verb\ NP \bullet PP$	[0,3]	Completer
	S35	$PP \rightarrow \bullet Prep\ NP$	[3,3]	Predictor
	S36	$S \rightarrow VP \bullet$	[0,3]	Completer

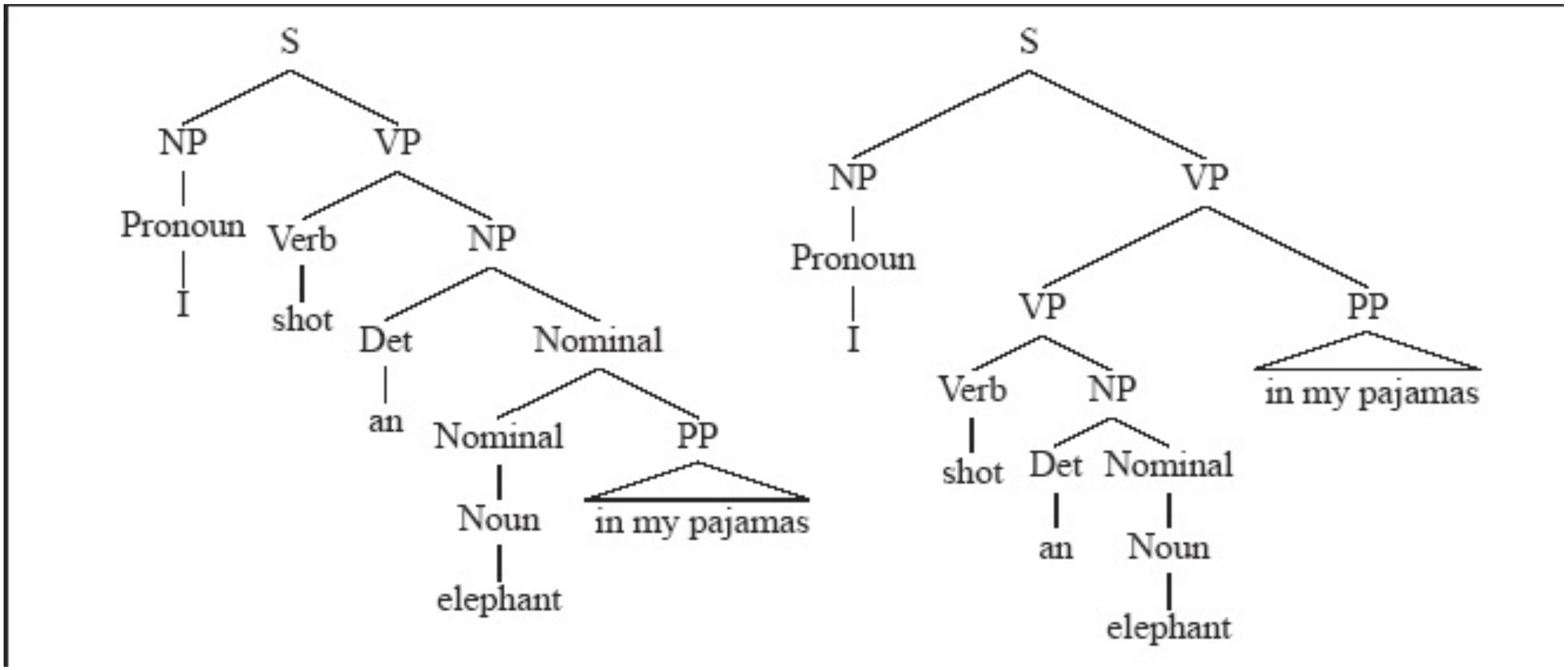
Details

- What kind of algorithms did we just describe (both Earley and CKY)
  - Not parsers – recognizers
    - The presence of an S state with the right attributes in the right place indicates a successful recognition.
    - But no parse tree… no parser
    - That's how we solve (not) an exponential problem in polynomial time

Back to Ambiguity

- Did we solve it?

Ambiguity


S
NP
VP
Pronoun
Verb
NP
I
shot
Det
Nominal
an
Nominal
PP
Noun
in my pajamas
elephant
S
NP
VP
Pronoun
VP
PP
I
Verb
NP
in my pajamas
shot
Det
Nominal
an
Noun
elephant

Converting Earley from Recognizer to Parser

- With the addition of a few pointers we have a parser
- Augment the “Completer” to point to where we came from.

Augmenting the chart with structural information

Step	Dotted rule	Span	Step	Backpointer
S8	Verb → book •	[0,1]	Scanner	
S9	VP → Verb •	[0,1]	Completer	S8
S10	S → VP •	[0,1]	Completer	S9
S11	VP → Verb • NP	[0,1]	Completer	S8
S12	NP → • Det Nom	[1,1]	Predictor	S11
S13	NP → • PropN	[1,1]	Predictor	S11

Retrieving Parse Trees from Chart

- All the possible parses for an input are in the table
- We just need to read off all the backpointers from every complete S in the last column of the table
- Find all the S -> X . [0,N+1]
- Follow the structural traces from the Completer
- Of course, this won't be polynomial time, since there could be an exponential number of trees
- So we can at least represent ambiguity efficiently

How to do parse disambiguation

- Probabilistic methods
- Augment the grammar with probabilities
- Then modify the parser to keep only most probable parses
- And at the end, return the most probable parse