CS 114
Introduction to Computational Linguistics

Grammar and Parsing (II)

February 8, 2008

James Pustejovsky

Thanks to Dan Jurafsky and Jim Martin for many of these slides!
www.cs.brandeis.edu/~cs114/slides/114.08.lec10.ppt
www.stanford.edu/class/linguist180/180.07.lec10.ppt

Grammars and Parsing

- Context-Free Grammars and Constituency
- Some common CFG phenomena for English
- Baby parsers: Top-down and Bottom-up Parsing
- Today: Real parsers: Dynamic Programming parsing
  - CKY
- Probabilistic parsing
- Optional section: the Earley algorithm

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

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

Sample Grammar

S → *NP VP*
S → *Aux NP VP*
S → *VP*
NP → *Pronoun*
NP → *Proper-Noun*
NP → *Det Nominal*
Nominal → *Noun*
Nominal → *Nominal Noun*
Nominal → *Nominal PP*
VP → *Verb*
VP → *Verb NP*
VP → *Verb NP PP*
VP → *Verb PP*
VP → *VP PP*
PP → *Preposition NP*

Det → *that* | *this* | *a*
Noun → *book* | *flight* | *meal* | *money*
Verb → *book* | *include* | *prefer*
Pronoun → *I* | *she* | *me*
Proper-Noun → *Houston* | *TWA*
Aux → *does*
Preposition → *from* | *to* | *on* | *near* | *through*

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.

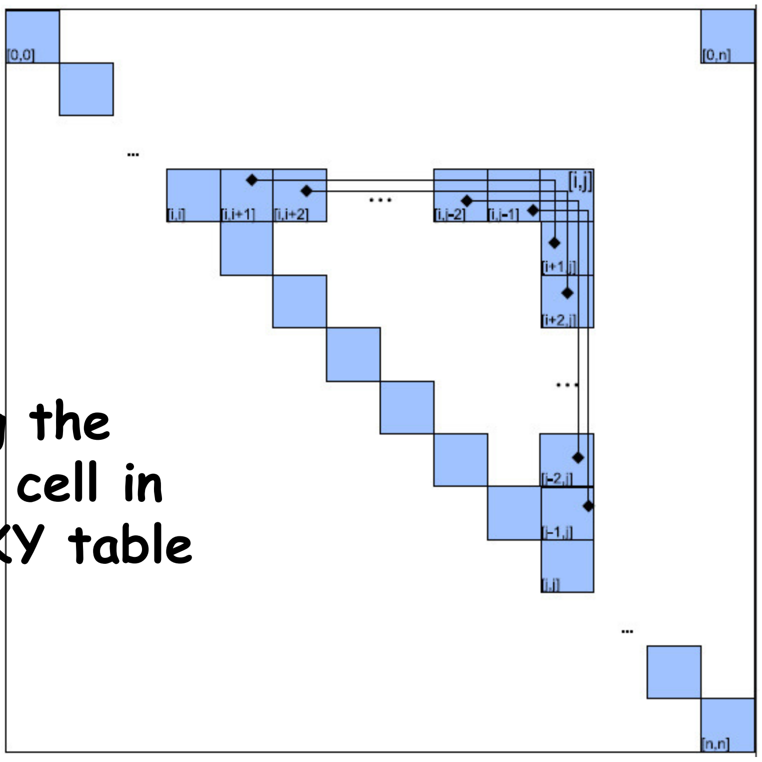
CKY

- 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

CKY Algorithm

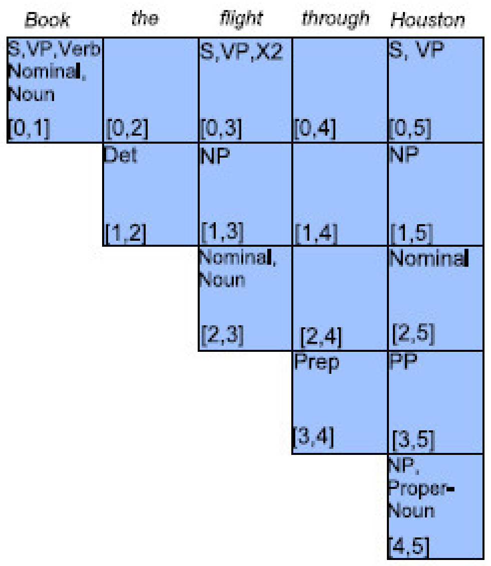
```
function CKY-PARSE(words, grammar) returns table

  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
              $table[i,j] \leftarrow table[i,j] \cup$
                          $\{A \mid A \rightarrow BC \in grammar,$
                              $B \in table[i,k],$
                              $C \in table[k,j]\}$
```

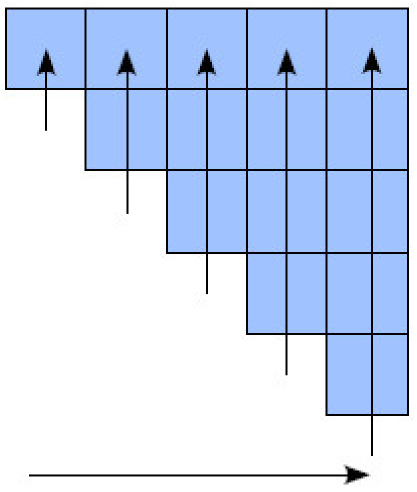
Note

- 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]

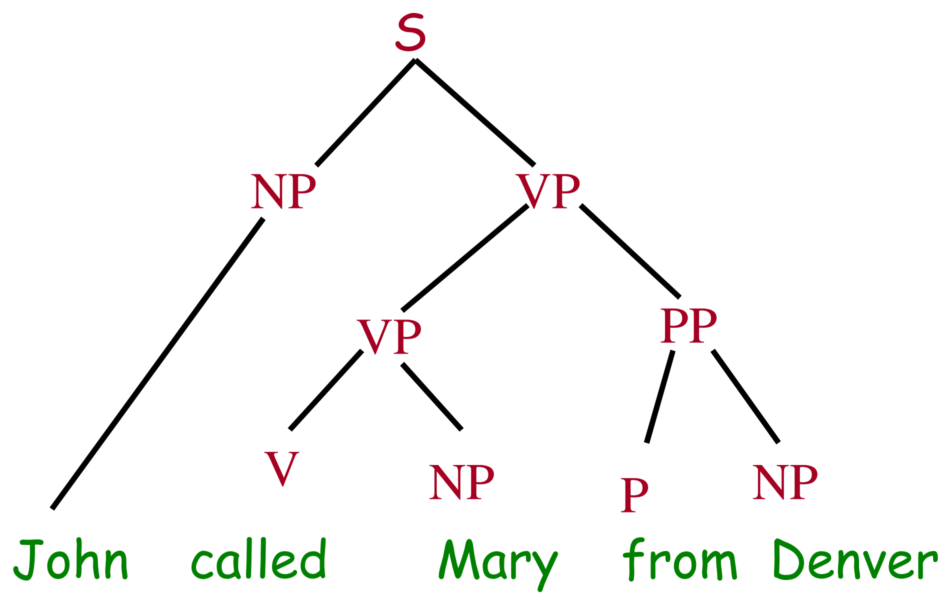


CYK Example

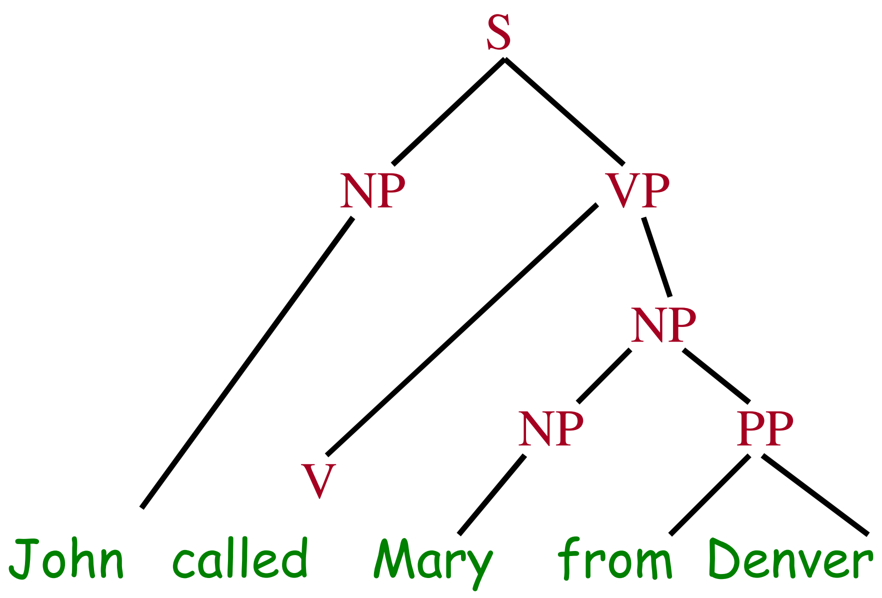
- S -> NP VP
- VP -> V NP
- NP -> NP PP
- VP -> VP PP
- PP -> P NP

- NP -> John, Mary, Denver
- V -> called
- P -> from

Example



Example



Example

				NP
			P	Denver
		NP	from	
	V	Mary		
NP	called			
John				

Example

				NP
			P	Denver
		NP	from	
X	V	Mary		
NP	called			
John				

Example

				NP
			P	Denver
	VP $\rightarrow$ $\downarrow$	NP	from	
X	V	Mary		
NP	called			
John				

Example

				NP
		X	P	Denver
	VP	NP	from	
X	V	Mary		
NP	called			
John				

Example

			PP	NP
		X	P	Denver
	VP	NP	from	
X	V	Mary		
NP	called			
John				

Example

			PP	NP
		X	P	Denver
S	VP	NP	from	
	V	Mary		
NP	called			
John				

Example

			PP	NP
	X	X	P	Denver
S	VP	NP	from	
X	V	Mary		
NP	called			
John				

Example

		NP	PP	NP
	X		P	Denver
S	VP	NP	from	
X	V	Mary		
NP	called			
John				

Example

		NP	PP	NP
X	X	X	P	Denver
S	VP	NP	from	
X	V	Mary		
NP	called			
John				

Example

	VP	NP	PP	NP
X	X	X	P	Denver
S	VP	NP	from	
X	V	Mary		
NP	called			
John				

Example

	VP	NP	PP	NP
X	X	X	P	Denver
S	VP	NP	from	
X	V	Mary		
NP	called			
John				

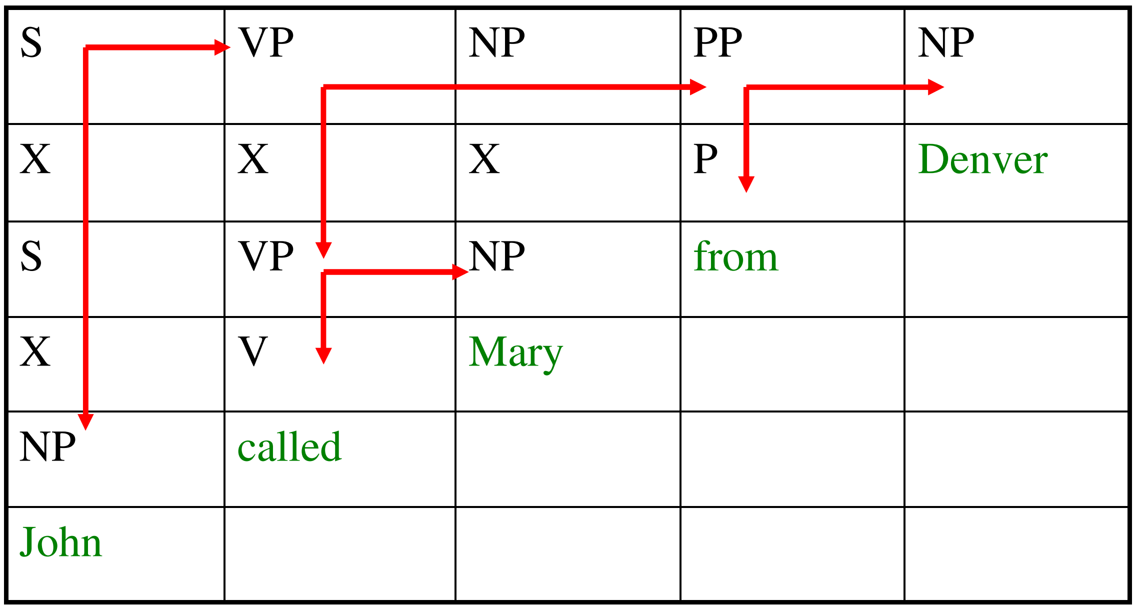
Example

	VP_1 VP_2	NP	PP	NP
X	X	X	P	Denver
S	VP	NP	from	
X	V	Mary		
NP	called			
John				

Example

S	VP_1 VP_2	NP	PP	NP
X	X	X	P	Denver
S	VP	NP	from	
X	V	Mary		
NP	called			
John				

Example

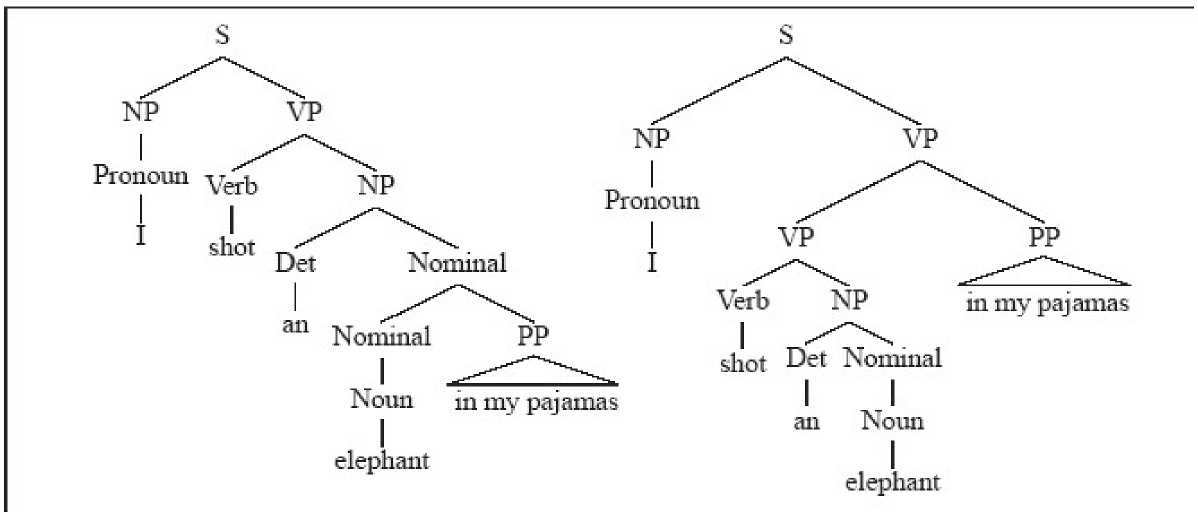


S	VP	NP	PP	NP
X	X	X	P	Denver
S	VP	NP	from	
X	V	Mary		
NP	called			
John				

Back to Ambiguity

- Did we solve it?

Ambiguity


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

Ambiguity

- 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.

Optional section: the Earley alg

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.

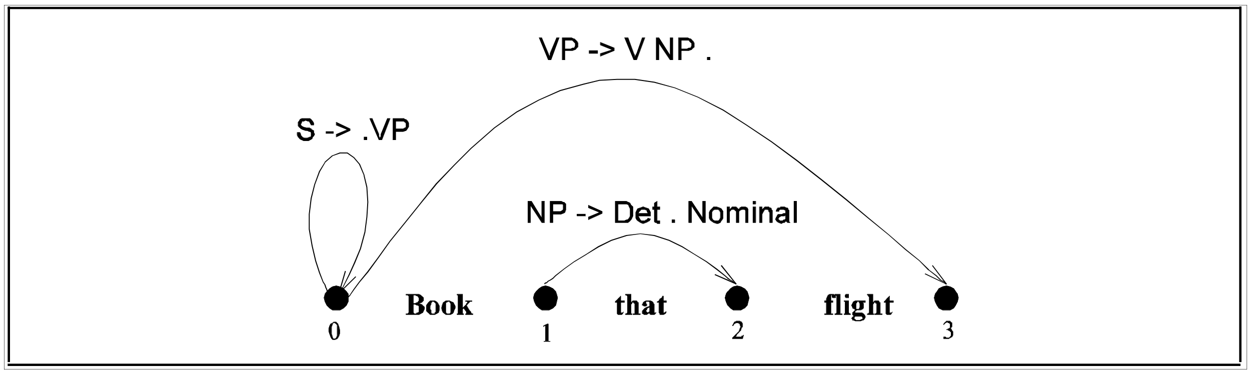
S -> · VP	A VP is predicted
NP -> Det · Nominal	An NP is in progress
VP -> V NP ·	A VP has been found

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

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 –> α · [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.

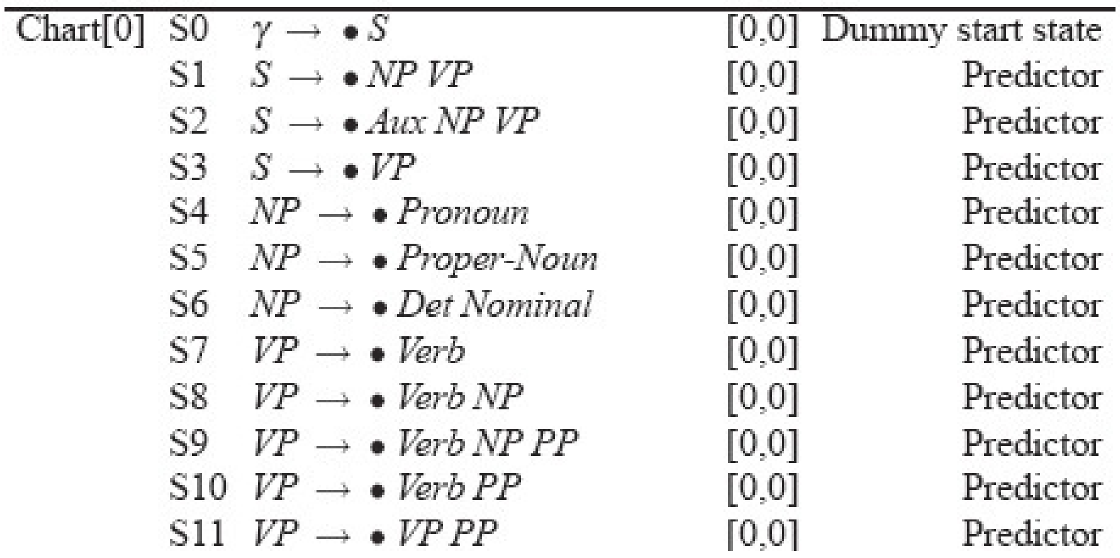
Earley

- 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

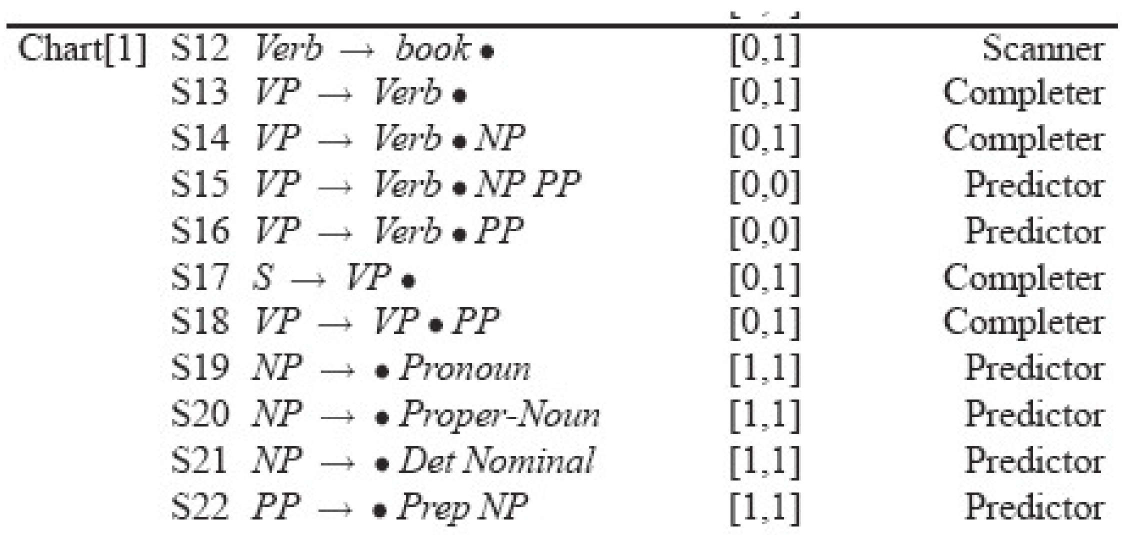
- Book that flight
- We should find… an S from 0 to 3 that is a completed state…

Example



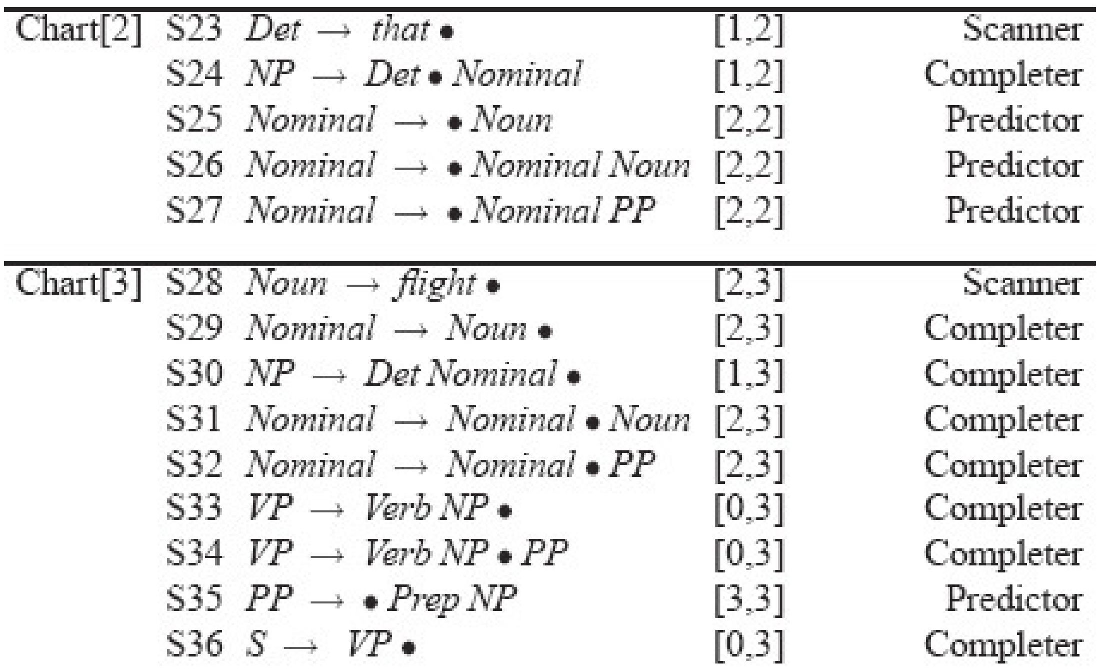
Chart[0]	S0	$\gamma \rightarrow \bullet S$	[0,0]	Dummy start state
	S1	$S \rightarrow \bullet NP\ VP$	[0,0]	Predictor
	S2	$S \rightarrow \bullet Aux\ NP\ VP$	[0,0]	Predictor
	S3	$S \rightarrow \bullet VP$	[0,0]	Predictor
	S4	$NP \rightarrow \bullet Pronoun$	[0,0]	Predictor
	S5	$NP \rightarrow \bullet Proper\text{-}Noun$	[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* → *that* •	[1,2]	Scanner
	S24	*NP* → *Det* • *Nominal*	[1,2]	Completer
	S25	*Nominal* → • *Noun*	[2,2]	Predictor
	S26	*Nominal* → • *Nominal Noun*	[2,2]	Predictor
	S27	*Nominal* → • *Nominal PP*	[2,2]	Predictor
Chart[3]	S28	*Noun* → *flight* •	[2,3]	Scanner
	S29	*Nominal* → *Noun* •	[2,3]	Completer
	S30	*NP* → *Det Nominal* •	[1,3]	Completer
	S31	*Nominal* → *Nominal* • *Noun*	[2,3]	Completer
	S32	*Nominal* → *Nominal* • *PP*	[2,3]	Completer
	S33	*VP* → *Verb NP* •	[0,3]	Completer
	S34	*VP* → *Verb NP* • *PP*	[0,3]	Completer
	S35	*PP* → • *Prep NP*	[3,3]	Predictor
	S36	*S* → *VP* •	[0,3]	Completer

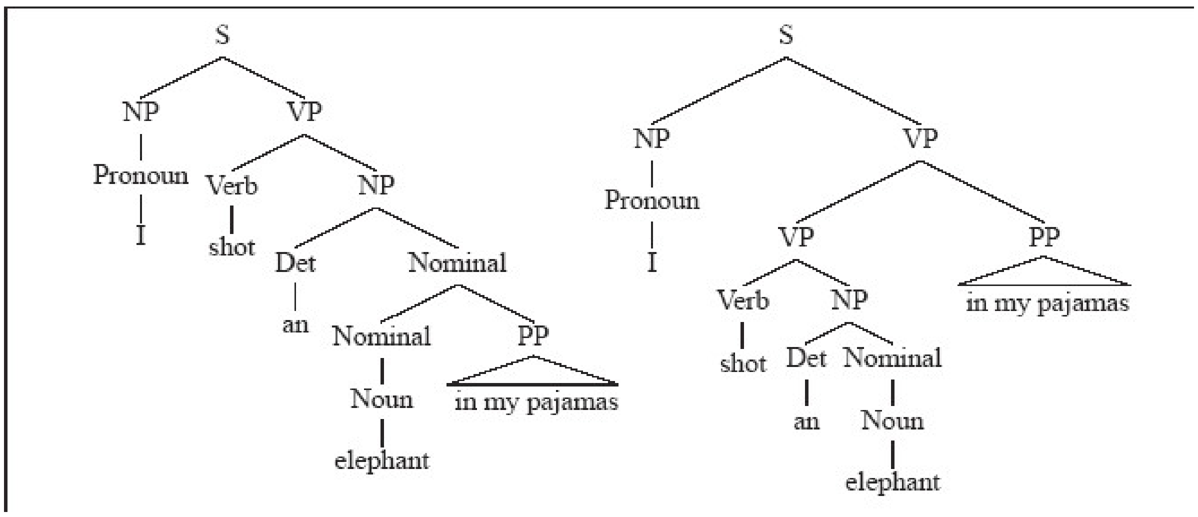
Details

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

Ambiguity

- 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 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

Chart[1]

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 NOMINAL*	[1,1]	Predictor	
S13	*NP* ■ ▪ *Proper-Noun*	[1,1]	Predictor	

Chart[2]

Det ■ *that*▪	[1,2]	Scanner
NP ■ *Det*▪*NOMINAL*	[1,2]	Completer
NOMINAL ■ ▪ *Noun*	[2,2]	Predictor
NOMINAL ■ ▪ *Noun NOMINAL*	[2,2]	Predictor

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

Probabilistic CFGs

- The probabilistic model
  - Assigning probabilities to parse trees
- Getting the probabilities for the model
- Parsing with probabilities
  - Slight modification to dynamic programming approach
  - Task is to find the max probability tree for an input

Probability Model

- Attach probabilities to grammar rules
- The expansions for a given non-terminal sum to 1

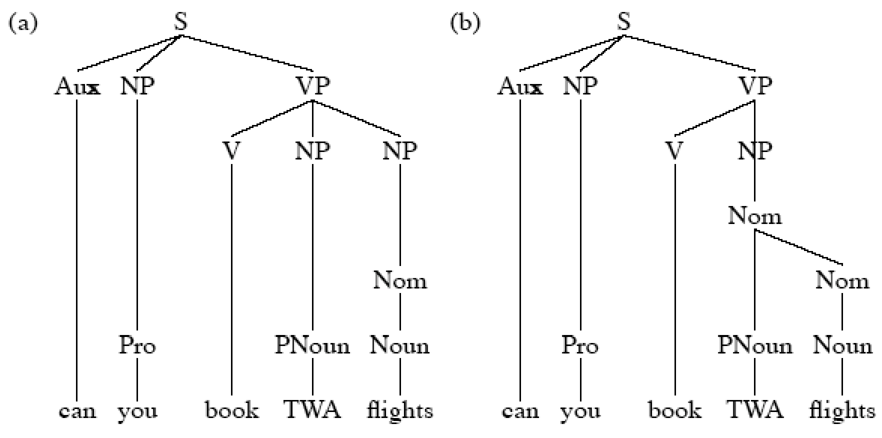
VP -> Verb	.55
VP -> Verb NP	.40
VP -> Verb NP NP	.05

  - Read this as P(Specific rule | LHS)

PCFG

S → *NP VP*	[.80]	*Det* → *that* [.05] | *the* [.80] | *a*	[.15]
S → *Aux NP VP*	[.15]	*Noun* → *book*	[.10]
S → *VP*	[.05]	*Noun* → *flights*	[.50]
NP → *Det Nom*	[.20]	*Noun* → *meal*	[.40]
NP → *Proper-Noun*	[.35]	*Verb* → *book*	[.30]
NP → *Nom*	[.05]	*Verb* → *include*	[.30]
NP → *Pronoun*	[.40]	*Verb* → *want*	[.40]
Nom → *Noun*	[.75]	*Aux* → *can*	[.40]
Nom → *Noun Nom*	[.20]	*Aux* → *does*	[.30]
Nom → *Proper-Noun Nom*	[.05]	*Aux* → *do*	[.30]
VP → *Verb*	[.55]	*Proper-Noun* → *TWA*	[.40]
VP → *Verb NP*	[.40]	*Proper-Noun* → *Denver*	[.40]
VP → *Verb NP NP*	[.05]	*Pronoun* → *you* [.40] | *I* [.60]	

PCFG



Rules			P	Rules			P
S	→	Aux NP VP	.15	S	→	Aux NP VP	.15
NP	→	Pro	.40	NP	→	Pro	.40
VP	→	V NP NP	.05	VP	→	V NP	.40
NP	→	Nom	.05	NP	→	Nom	.05
NP	→	PNoun	.35	Nom	→	PNoun Nom	.05
Nom	→	Noun	.75	Nom	→	Noun	.75
Aux	→	Can	.40	Aux	→	Can	.40
NP	→	Pro	.40	NP	→	Pro	.40
Pro	→	you	.40	Pro	→	you	.40
Verb	→	book	.30	Verb	→	book	.30
PNoun	→	TWA	.40	Pnoun	→	TWA	.40
Noun	→	flights	.50	Noun	→	flights	.50

Probability Model (1)

- A derivation (tree) consists of the set of grammar rules that are in the tree
- The probability of a tree is just the product of the probabilities of the rules in the derivation.

Probability model

$$P(T,S) = \prod_{n \in T} p(r_n)$$

- P(T,S) = P(T)P(S|T) = P(T); since P(S|T)=1

$$\begin{aligned} P(T_l) &= .15 * .40 * .05 * .05 * .35 * .75 * .40 * .40 * .40 \\ &\quad * .30 * .40 * .50 \\ &= 1.5 \times 10^{-6} \end{aligned}$$

$$\begin{aligned} P(T_r) &= .15 * .40 * .40 * .05 * .05 * .75 * .40 * .40 * .40 \\ &\quad * .30 * .40 * .50 \\ &= 1.7 \times 10^{-6} \end{aligned}$$

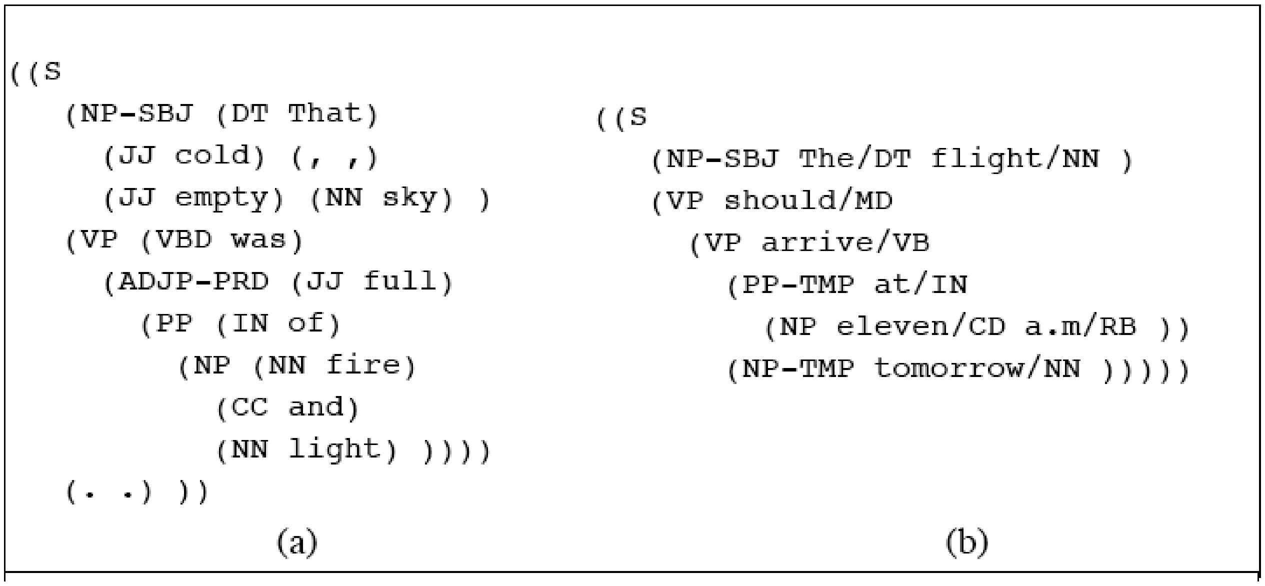
Probability Model (1.1)

- The probability of a word sequence P(S) is the probability of its tree in the unambiguous case.
- It’s the sum of the probabilities of the trees in the ambiguous case.

Getting the Probabilities

- From an annotated database (a treebank)
  - So for example, to get the probability for a particular VP rule just count all the times the rule is used and divide by the number of VPs overall.

TreeBanks



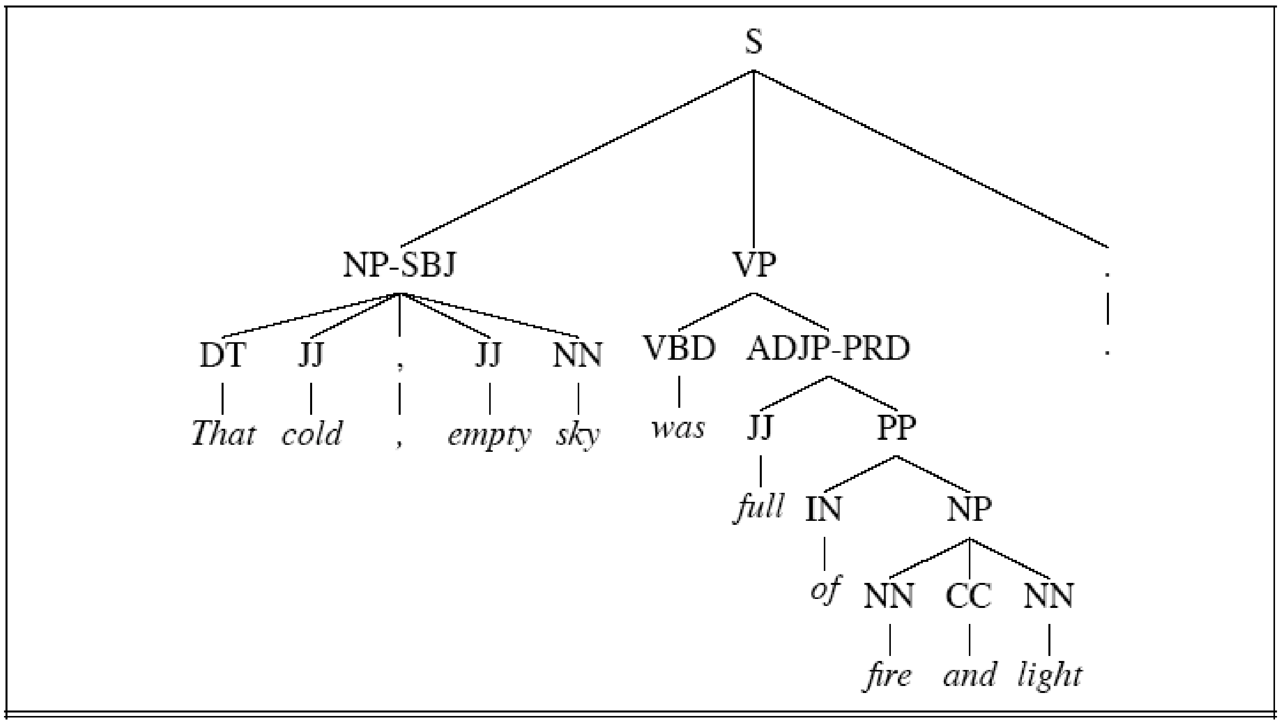
```
((S
   (NP-SBJ (DT That)
     (JJ cold) (, ,)
     (JJ empty) (NN sky) )
   (VP (VBD was)
     (ADJP-PRD (JJ full)
       (PP (IN of)
         (NP (NN fire)
           (CC and)
           (NN light) ))))
   (. .) ))
```

(a)

```
((S
   (NP-SBJ The/DT flight/NN )
   (VP should/MD
     (VP arrive/VB
       (PP-TMP at/IN
         (NP eleven/CD a.m/RB ))
       (NP-TMP tomorrow/NN )))))
```

(b)

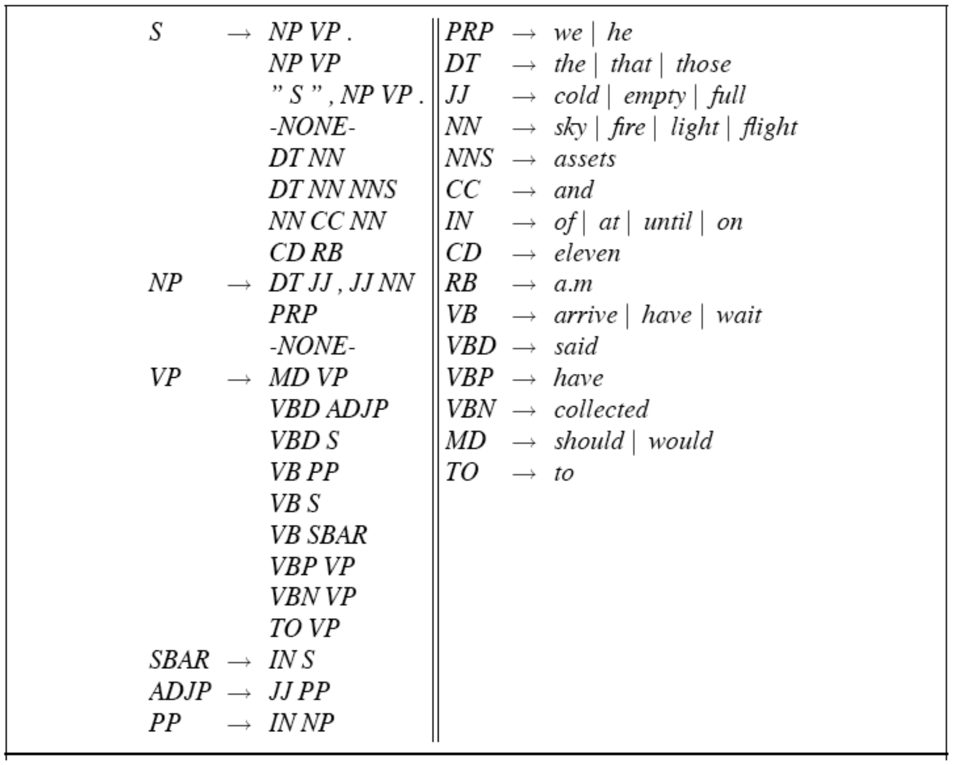
Treebanks


S
NP-SBJ
VP
.
DT
JJ
,
JJ
NN
VBD
ADJP-PRD
That
cold
,
empty
sky
was
JJ
PP
full
IN
NP
of
NN
CC
NN
fire
and
light
.

Treebanks

```
( (S (‘‘ ‘‘)
    (S-TPC-2
      (NP-SBJ-1 (PRP We) )
      (VP (MD would)
        (VP (VB have)
          (S
            (NP-SBJ (-NONE- *-1) )
            (VP (TO to)
              (VP (VB wait)
                (SBAR-TMP (IN until)
                  (S
                    (NP-SBJ (PRP we) )
                    (VP (VBP have)
                      (VP (VBN collected)
                        (PP-CLR (IN on)
                          (NP (DT those) (NNS assets) )))))))))))))
    (, ,) (’’ ’’)
    (NP-SBJ (PRP he) )
    (VP (VBD said)
      (S (-NONE- *T*-2) ))
    (. .) ))
```

Treebank Grammars



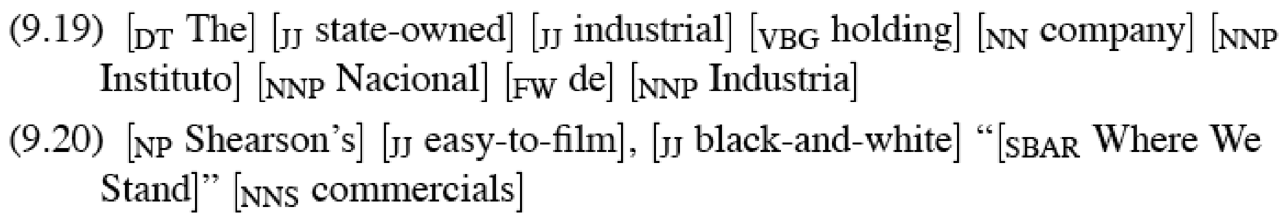
S	→	*NP VP .*	*PRP*	→	*we* | *he*
		NP VP	*DT*	→	*the* | *that* | *those*
		" S " , NP VP .	*JJ*	→	*cold* | *empty* | *full*
		-NONE-	*NN*	→	*sky* | *fire* | *light* | *flight*
		DT NN	*NNS*	→	*assets*
		DT NN NNS	*CC*	→	*and*
		NN CC NN	*IN*	→	*of* | *at* | *until* | *on*
		CD RB	*CD*	→	*eleven*
NP	→	*DT JJ , JJ NN*	*RB*	→	*a.m*
		PRP	*VB*	→	*arrive* | *have* | *wait*
		-NONE-	*VBD*	→	*said*
VP	→	*MD VP*	*VBP*	→	*have*
		VBD ADJP	*VBN*	→	*collected*
		VBD S	*MD*	→	*should* | *would*
		VB PP	*TO*	→	*to*
		VB S			
		VB SBAR			
		VBP VP			
		VBN VP			
		TO VP			
SBAR	→	*IN S*			
ADJP	→	*JJ PP*			
PP	→	*IN NP*			

Lots of flat rules

```
NP → DT JJ NN
NP → DT JJ NNS
NP → DT JJ NN NN
NP → DT JJ JJ NN
NP → DT JJ CD NNS
NP → RB DT JJ NN NN
NP → RB DT JJ JJ NNS
NP → DT JJ JJ NNP NNS
NP → DT NNP NNP NNP NNP JJ NN
NP → DT JJ NNP CC JJ JJ NN NNS
NP → RB DT JJS NN NN SBAR
NP → DT VBG JJ NNP NNP CC NNP
NP → DT JJ NNS , NNS CC NN NNS NN
NP → DT JJ JJ VBG NN NNP NNP FW NNP
NP → NP JJ , JJ ‘‘ SBAR ’’ NNS
```

Example sentences from those rules

- Total: over 17,000 different grammar rules in the 1-million word Treebank corpus



(9.19) [$_{\text{DT}}$ The] [$_{\text{JJ}}$ state-owned] [$_{\text{JJ}}$ industrial] [$_{\text{VBG}}$ holding] [$_{\text{NN}}$ company] [$_{\text{NNP}}$ Instituto] [$_{\text{NNP}}$ Nacional] [$_{\text{FW}}$ de] [$_{\text{NNP}}$ Industria]

(9.20) [$_{\text{NP}}$ Shearson's] [$_{\text{JJ}}$ easy-to-film], [$_{\text{JJ}}$ black-and-white] "[$_{\text{SBAR}}$ Where We Stand]" [$_{\text{NNS}}$ commercials]

Probabilistic Grammar Assumptions

- We're assuming that there is a grammar to be used to parse with.
- We're assuming the existence of a large robust dictionary with parts of speech
- We're assuming the ability to parse (i.e. a parser)
- Given all that… we can parse probabilistically

Typical Approach

- Bottom-up (CKY) dynamic programming approach
- Assign probabilities to constituents as they are completed and placed in the table
- Use the max probability for each constituent going up

What’s that last bullet mean?

- Say we’re talking about a final part of a parse
  - $S\text{->}{}_0NP_iVP_j$

The probability of the S is…
P(S->NP VP)*P(NP)*P(VP)

The green stuff is already known. We’re doing bottom-up parsing

Max

- I said the P(NP) is known.
- What if there are multiple NPs for the span of text in question (0 to i)?
- Take the max (where?)

Problems with PCFGs

- The probability model we're using is just based on the rules in the derivation…
  - Doesn't use the words in any real way
  - Doesn't take into account where in the derivation a rule is used

Solution

- Add lexical dependencies to the scheme…
  - Infiltrate the predilections of particular words into the probabilities in the derivation
  - I.e. Condition the rule probabilities on the actual words

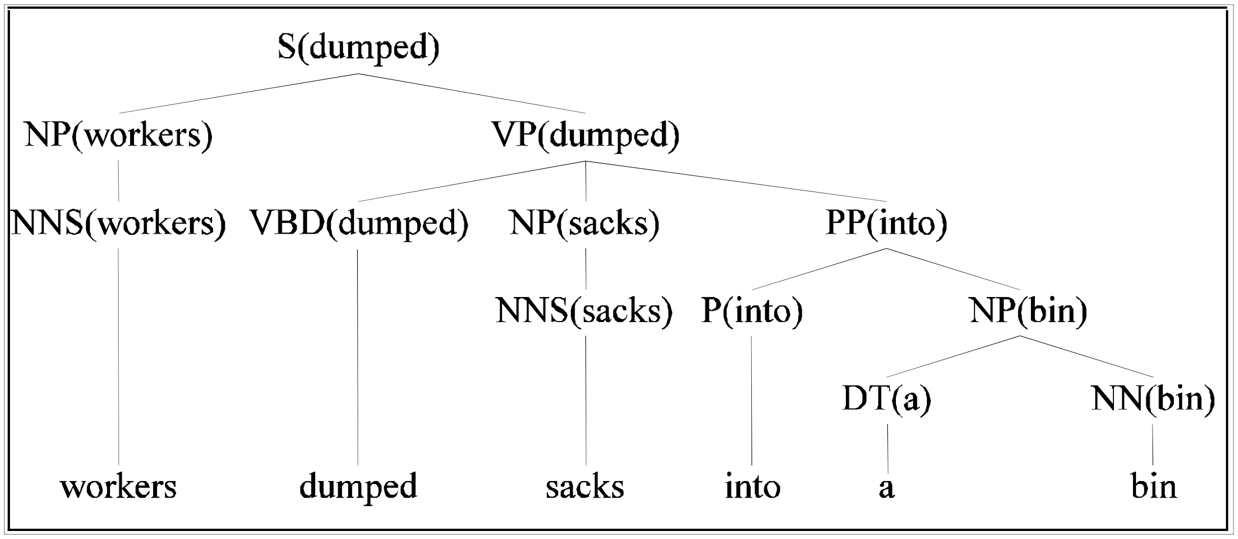
Heads

- To do that we're going to make use of the notion of the head of a phrase
  - The head of an NP is its noun
  - The head of a VP is its verb
  - The head of a PP is its preposition

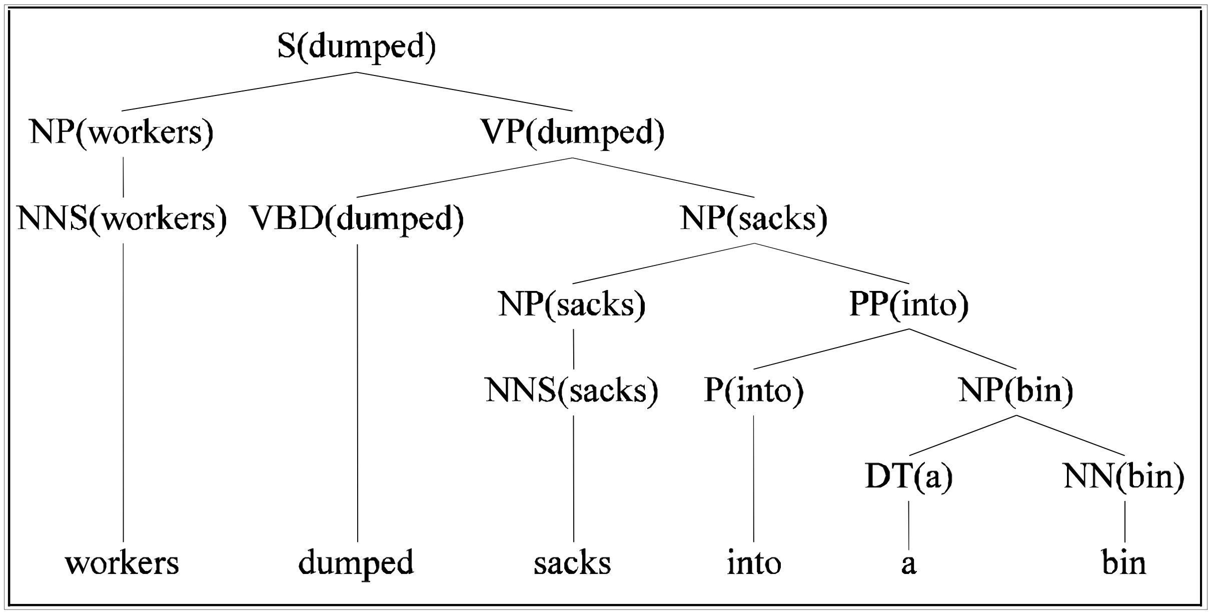
  (It's really more complicated than that but this will do.)

Example (right)

Attribute grammar



Example (wrong)



How?

- We used to have
  - VP -> V NP PP   P(rule|VP)
    - That's the count of this rule divided by the number of VPs in a treebank
- Now we have
  - VP(dumped)-> V(dumped) NP(sacks)PP(in)
  - P(r|VP ^ dumped is the verb ^ sacks is the head of the NP ^ in is the head of the PP)
  - Not likely to have significant counts in any treebank

Declare Independence

- When stuck, exploit independence and collect the statistics you can…
- We'll focus on capturing two things
  - Verb subcategorization
    - Particular verbs have affinities for particular VPs
  - Objects affinities for their predicates (mostly their mothers and grandmothers)
    - Some objects fit better with some predicates than others

Subcategorization

- Condition particular VP rules on their head… so

  r: VP -> V NP PP P(r|VP)

  Becomes

  P(r | VP ^ dumped)

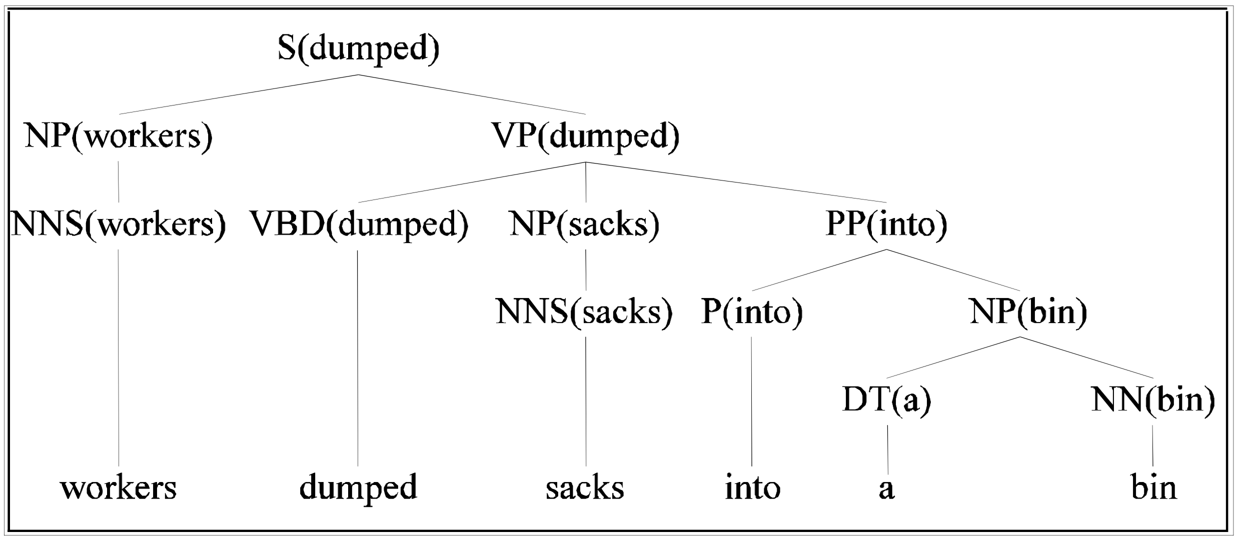
  What's the count?

  How many times was this rule used with (head) dump, divided by the number of VPs that dump appears (as head) in total

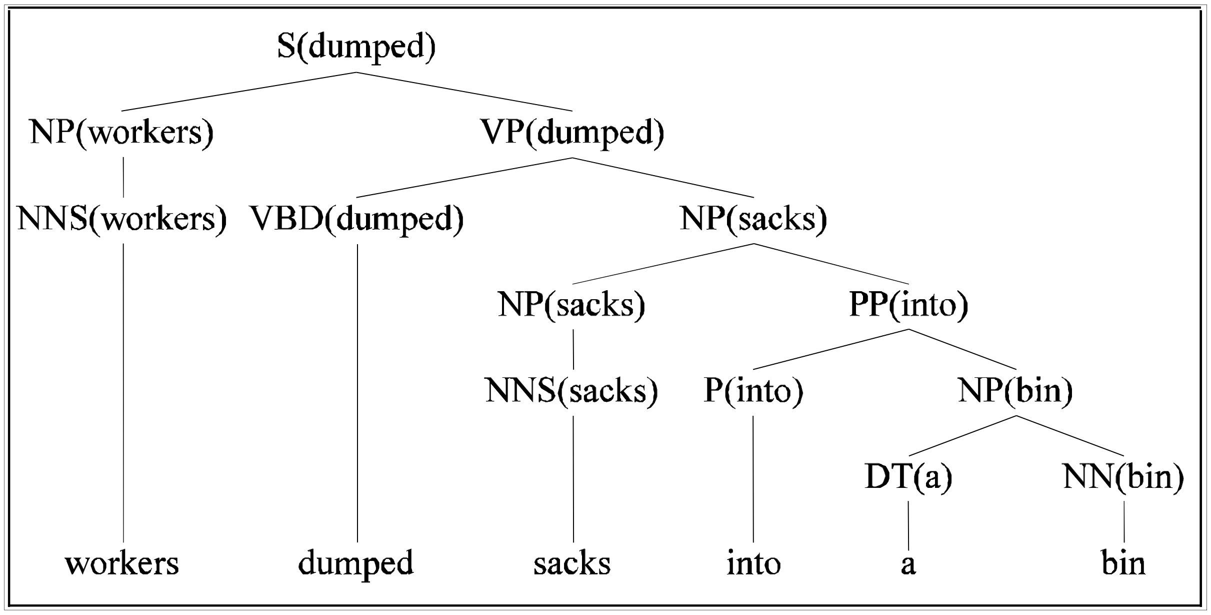
Preferences

- Subcat captures the affinity between VP heads (verbs) and the VP rules they go with.
- What about the affinity between VP heads and the heads of the other daughters of the VP
- Back to our examples…

Example (right)



Example (wrong)



Preferences

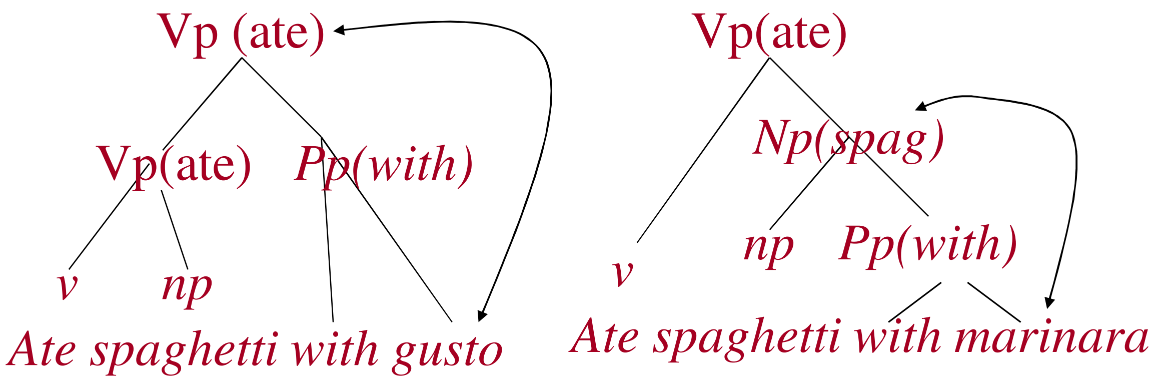
- The issue here is the attachment of the PP. So the affinities we care about are the ones between dumped and into vs. sacks and into.
- So count the places where dumped is the head of a constituent that has a PP daughter with into as its head and normalize
- Vs. the situation where sacks is a constituent with into as the head of a PP daughter.

Preferences (2)

- Consider the VPs
  - Ate spaghetti with gusto
  - Ate spaghetti with marinara
- The affinity of gusto for eat is much larger than its affinity for spaghetti
- On the other hand, the affinity of marinara for spaghetti is much higher than its affinity for ate

Preferences (2)

- Note the relationship here is more distant and doesn't involve a headword since gusto and marinara aren't the heads of the PPs.



Summary

- Context-Free Grammars
- Parsing
  - Top Down, Bottom Up Metaphors
  - Dynamic Programming Parsers: CKY. Earley
- Disambiguation:
  - PCFG
  - Probabilistic Augmentations to Parsers
  - Treebanks