Polynomial time parsing of PCFGs

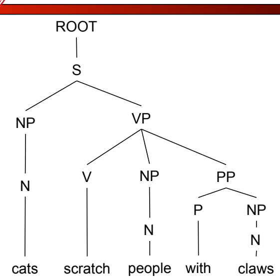
Christopher Manning

(some slides from Pi-Chuan Chang)

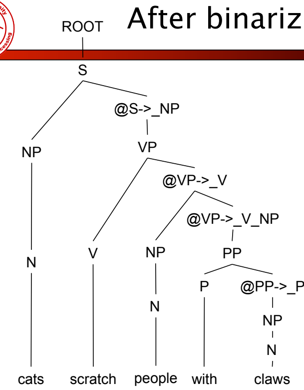
0. Chomsky Normal Form

- All rules are of the form X → Y Z or X → w.
- A transformation to this form doesn't change the weak generative capacity of CFGs.
  - With some extra book-keeping in symbol names, you can even reconstruct the same trees with a detransform
  - Unaries/empties are removed recursively
  - *n*-ary rules introduce new nonterminals ($n > 2$)
    - VP → V NP PP becomes VP → V @VP-V and @VP-V → NP PP
- In practice it's a pain
  - Reconstructing *n*-aries is easy
  - Reconstructing unaries can be trickier
- But it makes parsing easier/more efficient

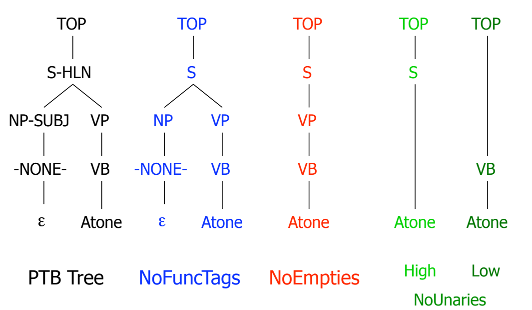
An example: before binarization…



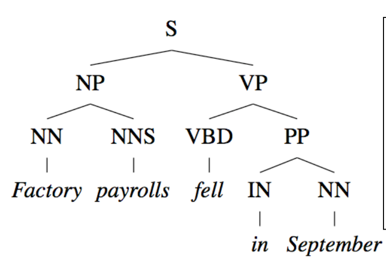
After binarization…



Treebank: empties and unaries



Constituency Parsing



PCFG

Rule Probs θ_i

θ_0: S → NP VP

θ_1: NP → NN NNS

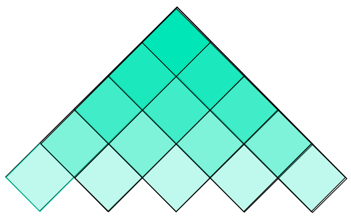
…

θ_{42}: NN→Factory

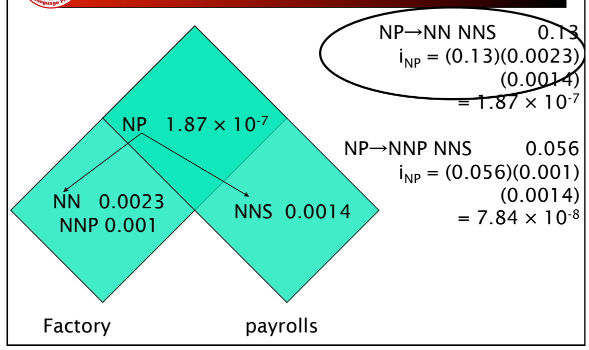
θ_{43}: NNS→payrolls

…

1. Cocke-Kasami-Younger (CKY) Constituency Parsing



Viterbi (Max) Scores



Extended CKY parsing

- Unaries can be incorporated into the algorithm
  - Messy, but doesn't increase algorithmic complexity
- Empties can be incorporated
  - Use fenceposts
  - Doesn't increase complexity; essentially like unaries
- Binarization is *vital*
  - Without binarization, you don't get parsing cubic in the length of the sentence
    - Binarization may be an explicit transformation or implicit in how the parser works (Early-style dotted rules), but it's always there.

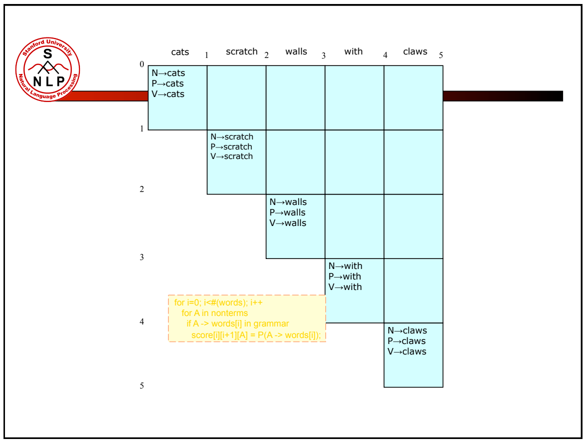
The CKY algorithm (1960/1965) … generalized

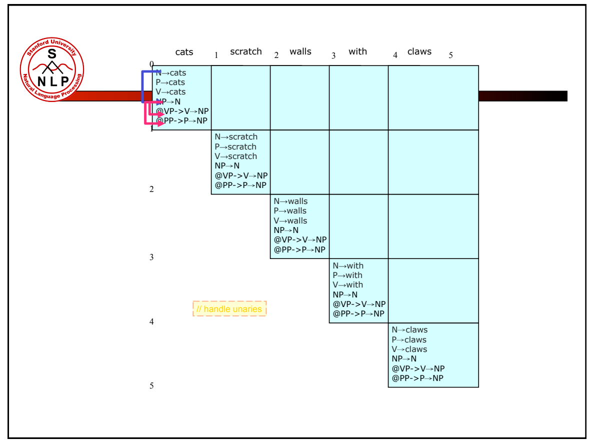
```
function CKY(words, grammar) returns most probable parse/prob
  score = new double[#(words)+1][#(words)+][#(nonterms)]
  back = new Pair[#(words)+1][#(words)+1][#nonterms]]
  for i=0; i<#(words); i++
    for A in nonterms
      if A -> words[i] in grammar
        score[i][i+1][A] = P(A -> words[i])
    //handle unaries
    boolean added = true
    while added
      added = false
      for A, B in nonterms
        if score[i][i+1][B] > 0 && A->B in grammar
          prob = P(A->B)*score[i][i+1][B]
          if(prob > score[i][i+1][A])
            score[i][i+1][A] = prob
            back[i][i+1] [A] = B
            added = true
```

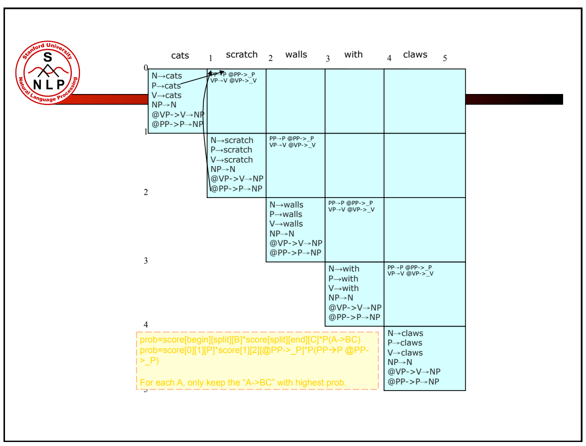
The CKY algorithm (1960/1965) … generalized

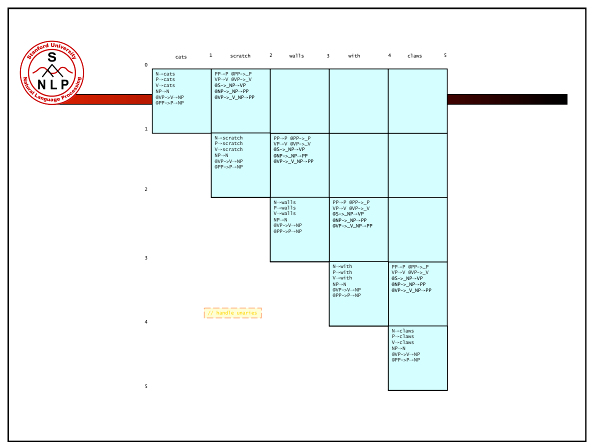
```
for span = 2 to #(words)
  for begin = 0 to #(words)- span
    end = begin + span
    for split = begin+1 to end-1
      for A,B,C in nonterms
        prob=score[begin][split][B]*score[split][end][C]*P(A->BC)
        if(prob > score[begin][end][A])
          score[begin]end][A] = prob
          back[begin][end][A] = new Triple(split,B,C)
      //handle unaries
      boolean added = true
      while added
        added = false
        for A, B in nonterms
          prob = P(A->B)*score[begin][end][B];
          if(prob > score[begin][end] [A])
            score[begin][end] [A] = prob
            back[begin][end] [A] = B
            added = true
return buildTree(score, back)
```

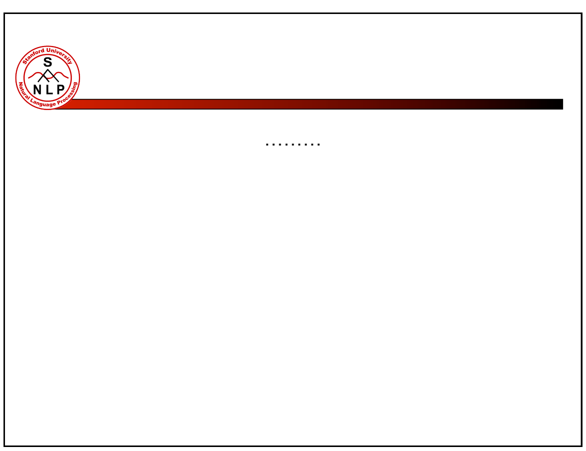
	cats 1	scratch 2	walls 3	with 4	claws 5
0	score[0][1]	score[0][2]	score[0][3]	score[0][4]	score[0][5]
1		score[1][2]	score[1][3]	score[1][4]	score[1][5]
2			score[2][3]	score[2][4]	score[2][5]
3				score[3][4]	score[3][5]
4					score[4][5]
5					

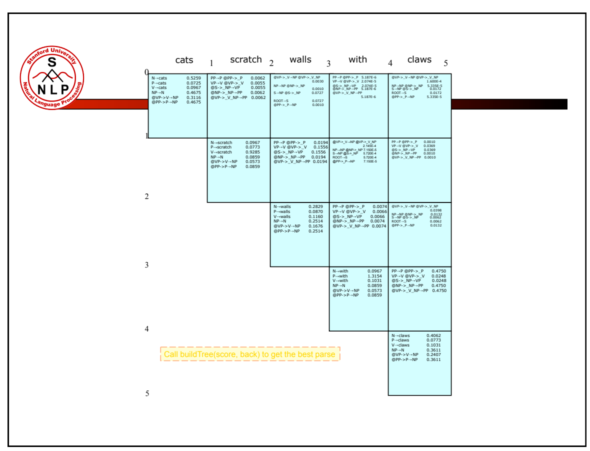

cats 1 scratch 2 walls 3 with 4 claws 5
N→cats
P→cats
V→cats
N→scratch
P→scratch
V→scratch
N→walls
P→walls
V→walls
N→with
P→with
V→with
N→claws
P→claws
V→claws
for i=0; i<#(words); i++
for A in nonterms
if A -> words[i] in grammar
score[i][i+1][A] = P(A -> words[i]);


cats 1 scratch 2 walls 3 with 4 claws 5
N→cats
P→cats
V→cats
NP→N
@VP->V→NP
@PP->P→NP
N→scratch
P→scratch
V→scratch
NP→N
@VP->V→NP
@PP->P→NP
N→walls
P→walls
V→walls
NP→N
@VP->V→NP
@PP->P→NP
N→with
P→with
V→with
NP→N
@VP->V→NP
@PP->P→NP
N→claws
P→claws
V→claws
NP→N
@VP->V→NP
@PP->P→NP
// handle unaries

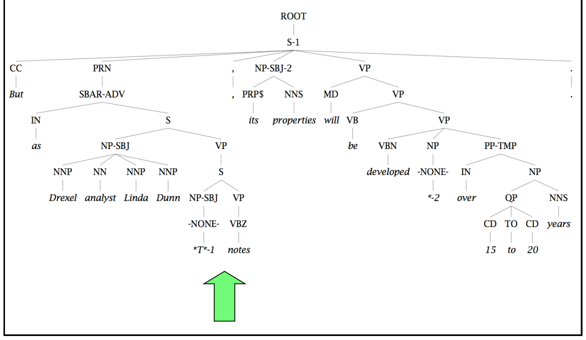

cats 1 scratch 2 walls 3 with 4 claws 5
N→cats
P→cats
V→cats
NP→N
@VP->V→NP
@PP->P→NP
PP→P @PP->_P
VP→V @VP->_V
N→scratch
P→scratch
V→scratch
NP→N
@VP->V→NP
@PP->P→NP
PP→P @PP->_P
VP→V @VP->_V
N→walls
P→walls
V→walls
NP→N
@VP->V→NP
@PP->P→NP
PP→P @PP->_P
VP→V @VP->_V
N→with
P→with
V→with
NP→N
@VP->V→NP
@PP->P→NP
PP→P @PP->_P
VP→V @VP->_V
N→claws
P→claws
V→claws
NP→N
@VP->V→NP
@PP->P→NP
prob=score[begin][split][B]*score[split][end][C]*P(A->BC)
prob=score[0][1][P]*score[1][2][@PP->_P]*P(PP->P @PP->_P)
For each A, only keep the "A->BC" with highest prob.


cats 1 scratch 2 walls 3 with 4 claws 5
N→cats
P→cats
V→cats
NP→N
@VP->V→NP
@PP->P→NP
PP→P @PP->_P
VP→V @VP->_V
@S->_NP→VP
@NP->_NP→PP
@VP->_V_NP→PP
N→scratch
P→scratch
V→scratch
NP→N
@VP->V→NP
@PP->P→NP
PP→P @PP->_P
VP→V @VP->_V
@S->_NP→VP
@NP->_NP→PP
@VP->_V_NP→PP
N→walls
P→walls
V→walls
NP→N
@VP->V→NP
@PP->P→NP
PP→P @PP->_P
VP→V @VP->_V
@S->_NP→VP
@NP->_NP→PP
@VP->_V_NP→PP
N→with
P→with
V→with
NP→N
@VP->V→NP
@PP->P→NP
PP→P @PP->_P
VP→V @VP->_V
@S->_NP→VP
@NP->_NP→PP
@VP->_V_NP→PP
// handle unaries

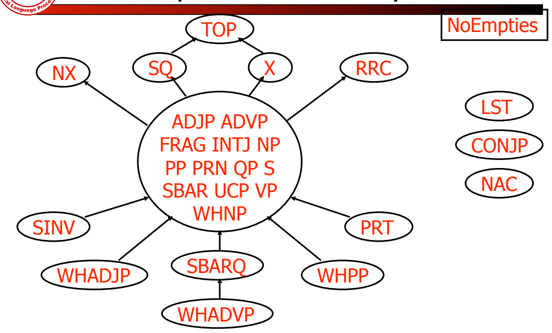

....


cats 1 scratch 2 walls 3 with 4 claws 5
N→cats 0.5259
P→cats 0.0725
V→cats 0.0967
NP→N 0.4675
@VP->V→NP 0.3116
@PP->P→NP 0.4675
PP→P @PP->_P 0.0062
VP→V @VP->_V 0.0055
@S->_NP→VP 0.0055
@NP->_NP→PP 0.0062
@VP->_V_NP→PP 0.0062
@VP->_V→NP @VP->_V_NP 0.0030
NP→NP @NP->_NP 0.0010
S→NP @S->_NP 0.0727
ROOT→S 0.0727
@PP->_P→NP 0.0010
PP→P @PP->_P 5.187E-6
VP→V @VP->_V 2.074E-5
@S->_NP→VP 2.074E-5
@NP->_NP→PP 5.187E-6
@VP->_V_NP→PP 5.187E-6
@VP->_V→NP @VP->_V_NP 1.600E-4
NP→NP @NP->_NP 5.335E-5
S→NP @S->_NP 0.0172
ROOT→S 0.0172
@PP->_P→NP 5.335E-5
N→scratch 0.0967
P→scratch 0.0773
V→scratch 0.9285
NP→N 0.0859
@VP->V→NP 0.0573
@PP->P→NP 0.0859
PP→P @PP->_P 0.0194
VP→V @VP->_V 0.1556
@S->_NP→VP 0.1556
@NP->_NP→PP 0.0194
@VP->_V_NP→PP 0.0194
@VP->_V→NP @VP->_V_NP 2.145E-4
NP→NP @NP->_NP 7.150E-5
S→NP @S->_NP 5.720E-4
ROOT→S 5.720E-4
@PP->_P→NP 7.150E-5
PP→P @PP->_P 0.0010
VP→V @VP->_V 0.0369
@S->_NP→VP 0.0369
@NP->_NP→PP 0.0010
@VP->_V_NP→PP 0.0010
N→walls 0.2829
P→walls 0.0870
V→walls 0.1160
NP→N 0.2514
@VP->V→NP 0.1676
@PP->P→NP 0.2514
PP→P @PP->_P 0.0074
VP→V @VP->_V 0.0066
@S->_NP→VP 0.0066
@NP->_NP→PP 0.0074
@VP->_V_NP→PP 0.0074
@VP->_V→NP @VP->_V_NP 0.0398
NP→NP @NP->_NP 0.0132
S→NP @S->_NP 0.0062
ROOT→S 0.0062
@PP->_P→NP 0.0132
N→with 0.0967
P→with 1.3154
V→with 0.1031
NP→N 0.0859
@VP->V→NP 0.0573
@PP->P→NP 0.0859
PP→P @PP->_P 0.4750
VP→V @VP->_V 0.0248
@S->_NP→VP 0.0248
@NP->_NP→PP 0.4750
@VP->_V_NP→PP 0.4750
N→claws 0.4062
P→claws 0.0773
V→claws 0.1031
NP→N 0.3611
@VP->V→NP 0.2407
@PP->P→NP 0.3611
Call buildTree(score, back) to get the best parse

Unary rules: alchemy in the land of treebanks



Same-Span Reachability



Efficient CKY parsing

- CKY parsing can be made very fast (!), partly due to the simplicity of the structures used.
  - But that means a lot of the speed comes from engineering details
  - And a little from cleverer filtering
  - Store chart as (ragged) 3 dimensional array of float (log probabilities)
    - score[start][end][category]
      - For treebank grammars the load is high enough that you don't really gain from lists of things that were possible
      - 50 wds: (50x50)/2x(1000 to 20000)x4 bytes = 5-100MB for parse triangle. Large. (Can move to beam for span[i][j].)
  - Use int to represent categories/words (Index)

Efficient CKY parsing

- Provide efficient grammar/lexicon accessors:
  - E.g., return list of rules with this left child category
  - Iterate over left child, check for zero (Neg. inf.) prob of X:[i,j] (abort loop), otherwise get rules with X on left
  - Some X:[i,j] can be filtered based on the input string
    - Not enough space to complete a long flat rule?
    - No word in the string can be a CC?
      - Using a lexicon of possible POS for words gives a lot of constraint rather than allowing all POS for words
    - Cf. later discussion of figures-of-merit/A* heuristics

2. An alternative … memoization

- A recursive (CNF) parser:

```
bestParse(X,i,j,s)
  if (j==i+1)
      return X -> s[i]
  (X->Y Z, k) = argmax score(X-> Y Z) *
              bestScore(Y,i,k,s) * bestScore(Z,k,j,s)
  parse.parent = X
  parse.leftChild = bestParse(Y,i,k,s)
  parse.rightChild = bestParse(Z,k,j,s)
  return parse
```

An alternative … memoization

```
bestScore(X,i,j,s)
  if (j == i+1)
      return tagScore(X, s[i])
  else
      return max score(X -> Y Z) *
           bestScore(Y, i, k) * bestScore(Z,k,j)
```

- Call: bestParse(Start, 1, sent.length(), sent)
  - Will this parser work?
  - Memory/time requirements?

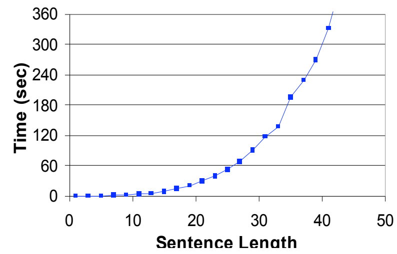
A memoized parser

- A simple change to record scores you know:

```
bestScore(X,i,j,s)
   if (scores[X][i][j] == null)
      if (j == i+1)
            score = tagScore(X, s[i])
      else
            score = max score(X -> Y Z) *
                    bestScore(Y, i, k) * bestScore(Z,k,j)
      scores[X][i][j] = score
   return scores[X][i][j]
```

- Memory and time complexity?

Runtime in practice: super-cubic!



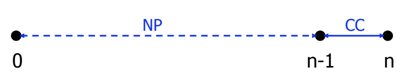
Best Fit Exponent:

3.47

Rule State Reachability

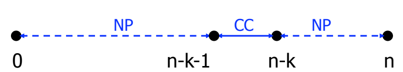
- Worse in practice because longer sentences "unlock" more of the grammar
- Many states are more likely to match larger spans!
- And because of various "systems" issues … cache misses, etc.

Example: NP CC . NP



1 Alignment

Example: NP CC NP . PP



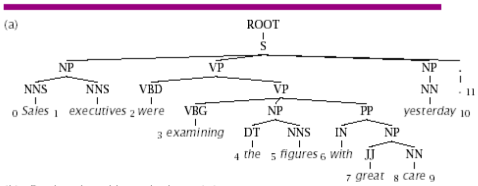
n Alignments

3. Evaluating Parsing Accuracy

- Most sentences are not given a completely correct parse by any currently existing parsers.
- Standardly for Penn Treebank parsing, evaluation is done in terms of the percentage of correct constituents (labeled spans).
- [label, start, finish]
- A constituent is a triple, all of which must be in the true parse for the constituent to be marked correct.

Evaluation

(a)



(b) Brackets in gold standard tree (a.):
S-(0:11), **NP-(0:2)**, VP-(2:9), VP-(3:9), **NP-(4:6)**, PP-(6-9), NP-(7,9), *NP-(9:10)

(c) Brackets in candidate parse:
S-(0:11), **NP-(0:2)**, VP-(2:10), VP-(3:10), NP-(4:10), **NP-(4:6)**, PP-(6-10), NP-(7,10)

(d)

Precision:	3/8 = 37.5%	Crossing Brackets:	0
Recall:	3/8 = 37.5%	Crossing Accuracy:	100%
Labeled Precision:	3/8 = 37.5%	Tagging Accuracy:	10/11 = 90.9%
Labeled Recall:	3/8 = 37.5%		

Evaluating Constituent Accuracy: LP/LR measure

- Let C be the number of correct constituents produced by the parser over the test set, M be the total number of constituents produced, and N be the total in the correct version [microaveraged]
- Precision = C/M
- Recall = C/N
- It is possible to artificially inflate either one.
- Thus people typically give the F-measure (harmonic mean) of the two. Not a big issue here; like average.
- This isn't necessarily a great measure … me and many other people think dependency accuracy would be better.

How good are PCFGs?

- Robust (usually admit everything, but with low probability)
- Partial solution for grammar ambiguity: a PCFG gives some idea of the plausibility of a sentence
- But not so good because the independence assumptions are too strong
- Give a probabilistic language model
  - But in a simple case it performs worse than a trigram model
- WSJ parsing accuracy: about 73% LP/LR F1
- The problem seems to be that PCFGs lack the lexicalization of a trigram model

Putting words into PCFGs

- A PCFG uses the actual words only to determine the probability of parts-of-speech (the preterminals)
- In many cases we need to know about words to choose a parse
- The head word of a phrase gives a good representation of the phrase's structure and meaning
  - Attachment ambiguities

    The astronomer saw the moon with the telescope
  - Coordination

    the dogs in the house and the cats
  - Subcategorization frames

    put versus *like*

(Head) Lexicalization

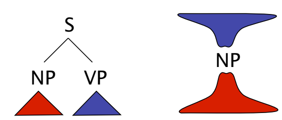
- *put* takes both an NP and a VP
  - *Sue put* [*the book*]$_{NP}$ [*on the table*]$_{PP}$
  - * *Sue put* [*the book*]$_{NP}$
  - * *Sue put* [*on the table*]$_{PP}$
- *like* usually takes an NP and not a PP
  - *Sue likes* [*the book*]$_{NP}$
  - * *Sue likes* [*on the table*]$_{PP}$
- We can't tell this if we just have a VP with a verb, but we can if we know *what* verb it is

4. Accurate Unlexicalized Parsing: PCFGs and Independence

- The symbols in a PCFG define independence assumptions:

  S → NP VP

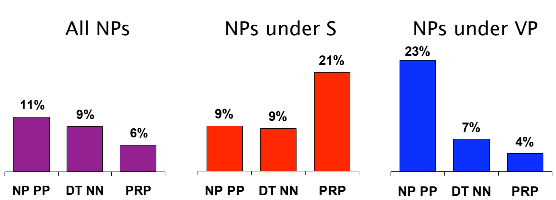
  NP → DT NN



  - At any node, the material inside that node is independent of the material outside that node, given the label of that node.
  - Any information that statistically connects behavior inside and outside a node must flow through that node.

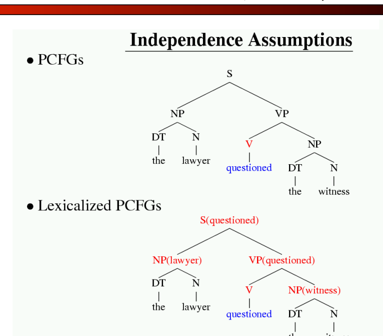
Non-Independence I

- Independence assumptions are often too strong.



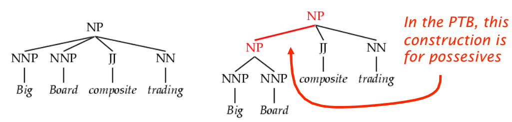
- Example: the expansion of an NP is highly dependent on the parent of the NP (i.e., subjects vs. objects).

Michael Collins (2003, COLT)



Non-Independence II

- Who cares?
  - NB, HMMs, all make false assumptions!
  - For generation/LMs, consequences would be obvious.
  - For parsing, does it impact accuracy?
- Symptoms of overly strong assumptions:
  - Rewrites get used where they don't belong.
  - Rewrites get used too often or too rarely.

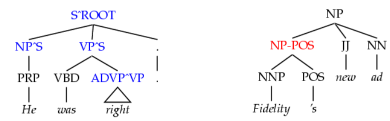


Breaking Up the Symbols

- We can relax independence assumptions by encoding dependencies into the PCFG symbols:

Parent annotation [Johnson 98]

Marking possesive NPs



- What are the most useful features to encode?

Annotations

- Annotations split the grammar categories into sub-categories.
- Conditioning on history vs. annotating
  - P(NP^S → PRP) is a lot like P(NP → PRP | S)
  - P(NP-POS → NNP POS) isn't history conditioning.
- Feature grammars vs. annotation
  - Can think of a symbol like NP^NP-POS as NP [parent:NP, +POS]
- After parsing with an annotated grammar, the annotations are then stripped for evaluation.

Experimental Setup

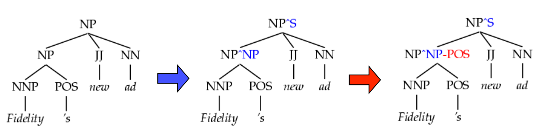
- Corpus: Penn Treebank, WSJ

Training:	sections	02-21
Development:	section	22 (first 20 files)
Test:	section	23

- Accuracy – F1: harmonic mean of per-node labeled precision and recall.
- Size – number of symbols in grammar.
  - Passive / complete symbols: NP, NP^S
  - Active / incomplete symbols: NP → NP CC •

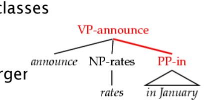
Experimental Process

- We'll take a highly conservative approach:
  - Annotate as sparingly as possible
  - Highest accuracy with fewest symbols
  - Error-driven, manual hill-climb, adding one annotation type at a time

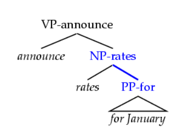


Lexicalization

- Lexical heads are important for certain classes of ambiguities (e.g., PP attachment):



- Lexicalizing grammar creates a much larger grammar.
  - Sophisticated smoothing needed
  - Smarter parsing algorithms needed
  - More data needed



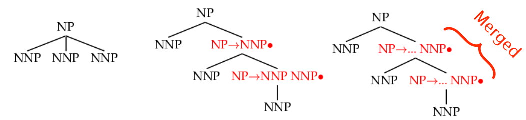
- How necessary is lexicalization?
  - Bilexical vs. monolexical selection
  - Closed vs. open class lexicalization

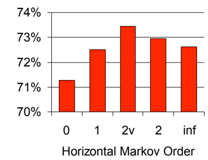
Unlexicalized PCFGs

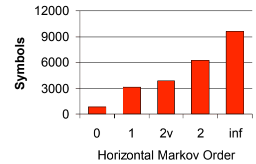
- What do we mean by an "unlexicalized" PCFG?
  - Grammar rules are not systematically specified down to the level of lexical items
    - NP-stocks is not allowed
    - NP^S-CC is fine
  - Closed vs. open class words (NP^S-the)
    - Long tradition in linguistics of using function words as features or markers for selection
    - Contrary to the bilexical idea of semantic heads
    - Open-class selection really a proxy for semantics
- Honesty checks:
  - Number of symbols: keep the grammar very small
  - No smoothing: over-annotating is a real danger

Horizontal Markovization

- Horizontal Markovization: Merges States

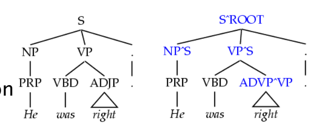


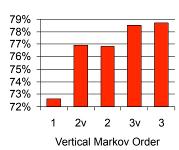


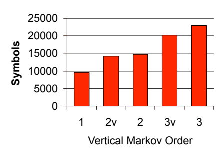


Vertical Markovization

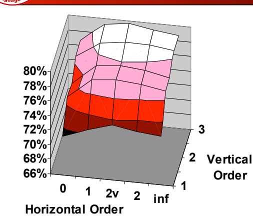
- Vertical Markov order: rewrites depend on past k ancestor nodes. (cf. parent annotation)

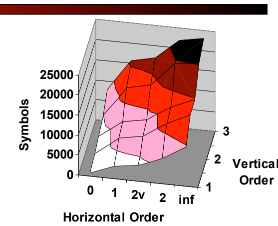






Vertical and Horizontal



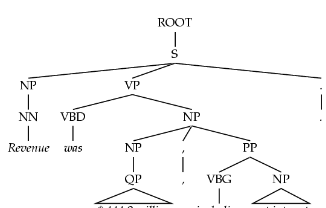


- Examples:
  - Raw treebank: v=1, h=∞
  - Johnson 98: v=2, h=∞
  - Collins 99: v=2, h=2
  - Best F1: v=3, h=2v

Model	F1	Size
Base: v=h=2v	77.8	7.5K

Unary Splits

- Problem: unary rewrites used to transmute categories so a high-probability rule can be used.

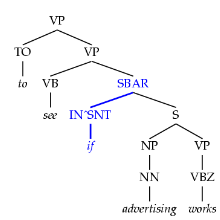


- Solution: Mark unary rewrite sites with -U

Annotation	F1	Size
Base	77.8	7.5K
UNARY	78.3	8.0K

Tag Splits

- Problem: Treebank tags are too coarse.
- Example: Sentential, PP, and other prepositions are all marked IN.
- Partial Solution:
  - Subdivide the IN tag.



Annotation	F1	Size
Previous	78.3	8.0K
SPLIT-IN	80.3	8.1K

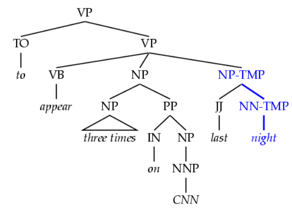
Other Tag Splits

- UNARY-DT: mark demonstratives as DT^U ("the X" vs. "those")
- UNARY-RB: mark phrasal adverbs as RB^U ("quickly" vs. "very")
- TAG-PA: mark tags with non-canonical parents ("not" is an RB^VP)
- SPLIT-AUX: mark auxiliary verbs with –AUX [cf. Charniak 97]
- SPLIT-CC: separate "but" and "&" from other conjunctions
- SPLIT-%: "%" gets its own tag.

F1	Size
80.4	8.1K
80.5	8.1K
81.2	8.5K
81.6	9.0K
81.7	9.1K
81.8	9.3K

Treebank Splits

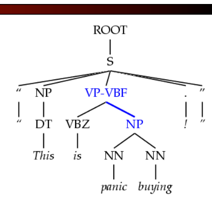
- The treebank comes with annotations (e.g., -LOC, -SUBJ, etc).
  - Whole set together hurt the baseline.
  - Some (-SUBJ) were less effective than our equivalents.
  - One in particular was very useful (NP-TMP) when pushed down to the head tag.
  - We marked gapped S nodes as well.



Annotation	F1	Size
Previous	81.8	9.3K
NP-TMP	82.2	9.6K
GAPPED-S	82.3	9.7K

Yield Splits

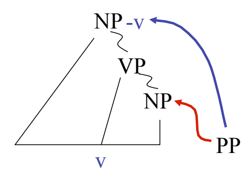
- Problem: sometimes the behavior of a category depends on something inside its future yield.
- Examples:
  - Possessive NPs
  - Finite vs. infinite VPs
  - Lexical heads!
- Solution: annotate future elements into nodes.



Annotation	F1	Size
Previous	82.3	9.7K
POSS-NP	83.1	9.8K
SPLIT-VP	85.7	10.5K

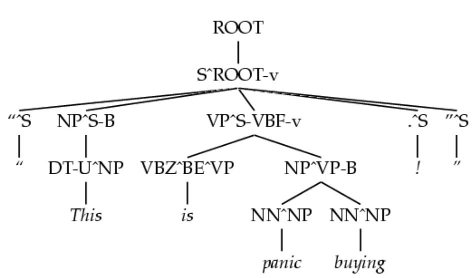
Distance / Recursion Splits

- Problem: vanilla PCFGs cannot distinguish attachment heights.
- Solution: mark a property of higher or lower sites:
  - Contains a verb.
  - Is (non)-recursive.
    - Base NPs [cf. Collins 99]
    - Right-recursive NPs



Annotation	F1	Size
Previous	85.7	10.5K
BASE-NP	86.0	11.7K
DOMINATES-V	86.9	14.1K
RIGHT-REC-NP	87.0	15.2K

A Fully Annotated Tree



Final Test Set Results

Parser	LP	LR	F1	CB	0 CB
Magerman 95	84.9	84.6	**84.7**	1.26	56.6
Collins 96	86.3	85.8	**86.0**	1.14	59.9
Klein & M 03	86.9	85.7	**86.3**	1.10	60.3
Charniak 97	87.4	87.5	**87.4**	1.00	62.1
Collins 99	88.7	88.6	**88.6**	0.90	67.1

- Beats "first generation" lexicalized parsers.