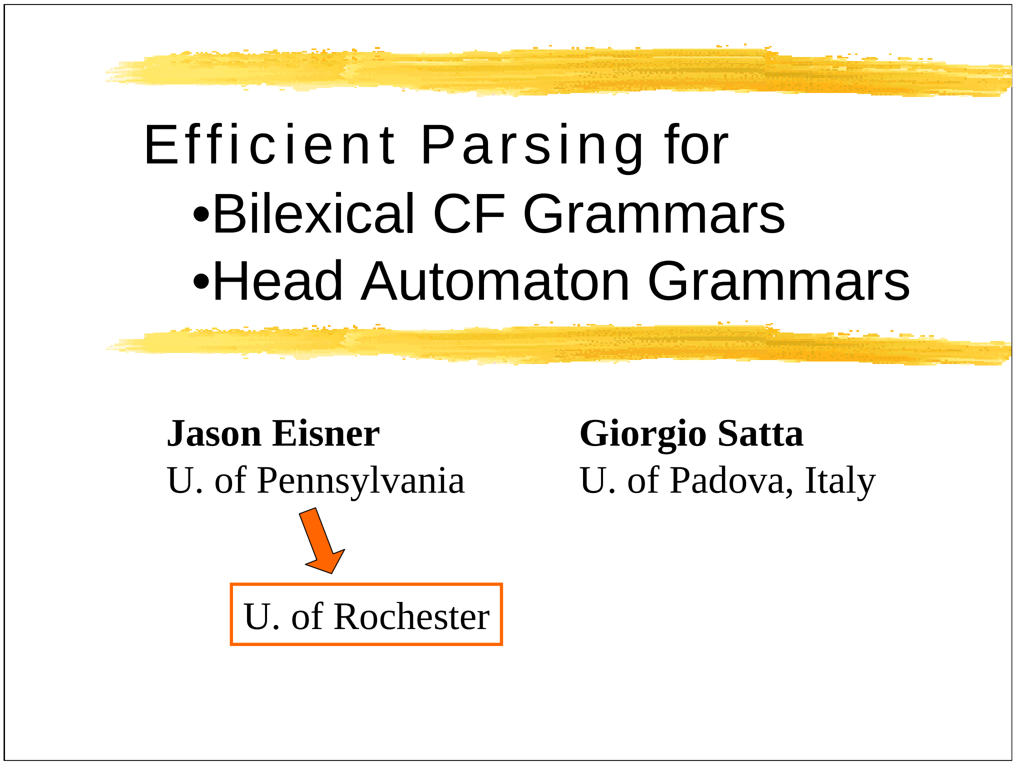


Efficient Parsing for

- Bilexical CF Grammars
- Head Automaton Grammars

Jason Eisner
U. of Pennsylvania

U. of Rochester

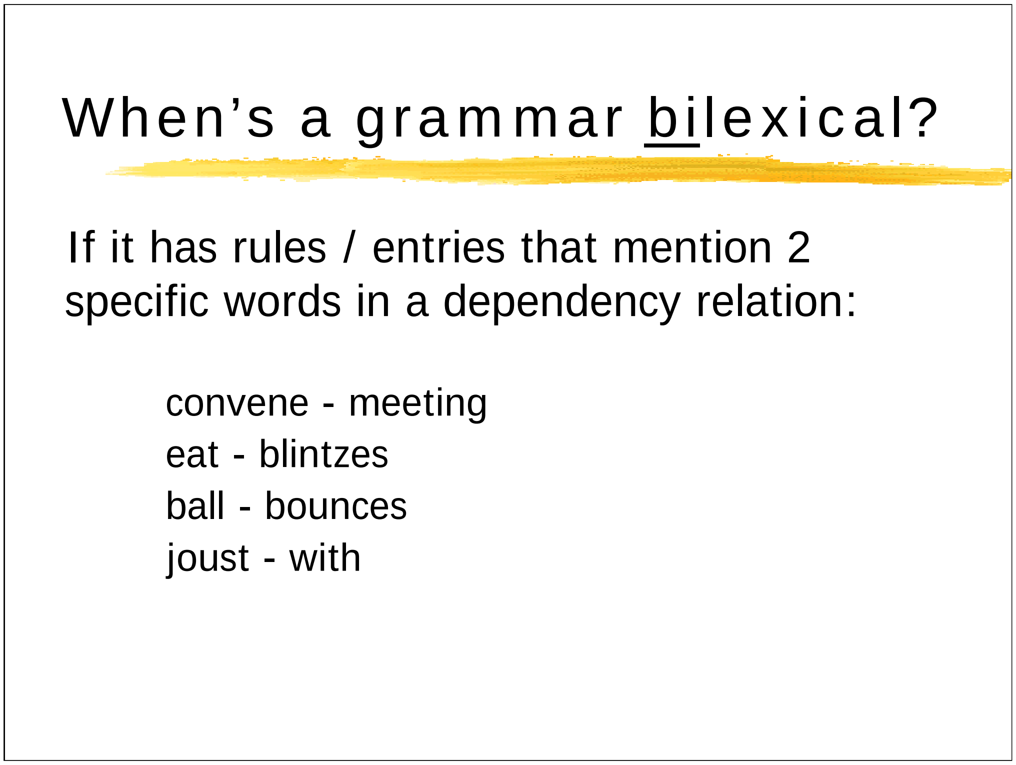
Giorgio Satta
U. of Padova, Italy

The speaker notes in this Powerpoint file may be helpful to the reader: they're Jason's approximate reconstruction (2 weeks later) of what he actually said in the talk. The slides themselves are the same except for a few minor changes; the most important change is that the performance graphs now have a few more points and also show formulas for their regression lines.

Hi, I'm JE and this work was an email collaboration with Giorgio Satta in Italy.

Giorgio's staying in Italy but asked me to send regards; I'm moving to Rochester to join all the compu- and psycho and pure linguists there.

I have to explain the title. Efficient parsing - that means it's an algorithms paper. For bilexical context-free grammars -- we know what a *lexicalized* grammar is, but when's a grammar *bilexical*?



When's a grammar bilexical?

If it has rules / entries that mention 2 specific words in a dependency relation:

convene - meeting
eat - blintzes
ball - bounces
joust - with

In English you can convene a meeting, but you can't convene a tete-a-tete, or a roundtable discussion, or a rendezvous, as far as I know.

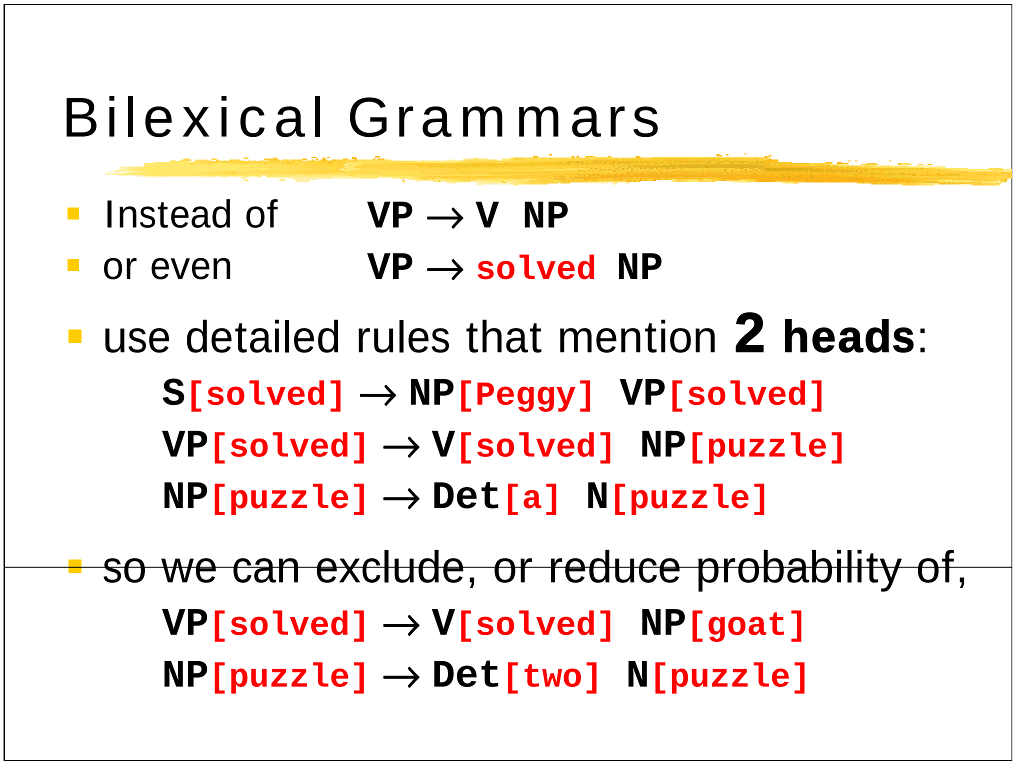
If you eat your words, corpus analysis suggests you're more likely to eat blintzes than blotters.

'Cause that's just the way the ball crumbles - I mean, bounces.

You never joust *against* your opponent, you joust *with* them, and so on.

Some of these collocations are genuinely lexical - i.e., they really are about the *words* - while some can be derived from semantic facts about the world. But whever these preferences come from, putting them in the grammar can help parsing.

Bear with me, while I introduce this earthshaking idea as if you'd never seen it before.



Bilexical Grammars

- Instead of VP → V NP
- or even VP → solved NP
- use detailed rules that mention **2 heads**:

  S[solved] → NP[Peggy] VP[solved]

  VP[solved] → V[solved] NP[puzzle]

  NP[puzzle] → Det[a] N[puzzle]

- so we can exclude, or reduce probability of,

  VP[solved] → V[solved] NP[goat]

  NP[puzzle] → Det[two] N[puzzle]

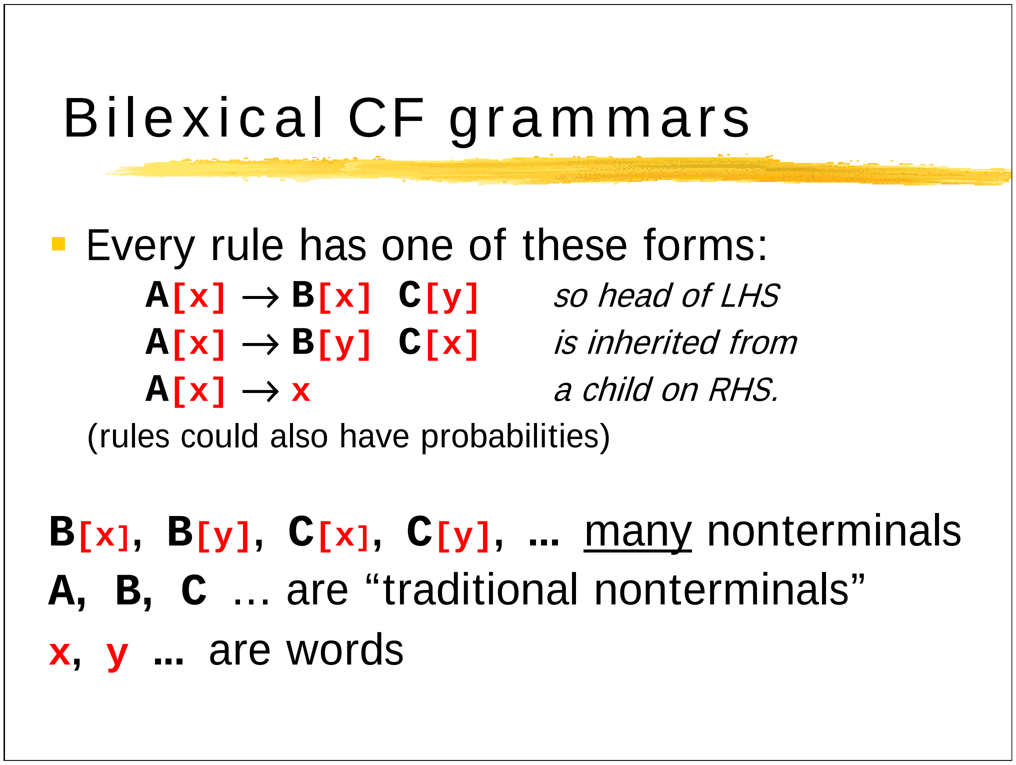
Here's an imperfect CF rule - imperfect because it just talks about verbs, V, as if they were all the same. But a verb can go in this rule only to the extent that it likes to be transitive. Some verbs hate to be transitive.

So we could lexicalize at the real transitive verbs, like *solved*. But how about a verb like *walk*? It's usually intransitive, but it can take a limited range of objects: you can walk the dog, maybe you can walk the cat, you can walk your yo-yo, you can walk the plank, you can walk the streets, and you can walk the walk (if you're willing to talk the talk).

So let's go one step further and list those objects. If each rule mentions 2 headwords, here's a very small fragment of a very large grammar:

An S headed by *solved* can be an NP with a nice animate, intelligent head, like Peggy, plus a VP headed by *solved*. That VP can be a V headed by *solved* plus an NP object appropriately headed by *puzzle*. Whaddya solve? Puzzles! And this puzzle-NP can take a singular determiner, *a*, which is another way of saying *puzzle* is a singular noun.

Those are good rules - they let us derive *Peggy solved a puzzle*. They're much better than these rules - *Peggy solved a goat*, or *two puzzle* - which, if they're in the grammar at all, should have much lower probability. And since we've made them separate rules, we can give them much lower probability.



Bilexical CF grammars

- Every rule has one of these forms:

  A[x] → B[x] C[y] *so head of LHS*
  A[x] → B[y] C[x] *is inherited from*
  A[x] → x *a child on RHS.*

  (rules could also have probabilities)

B[x], B[y], C[x], C[y], … <u>many</u> nonterminals

A, B, C … are “traditional nonterminals”

x, y … are words

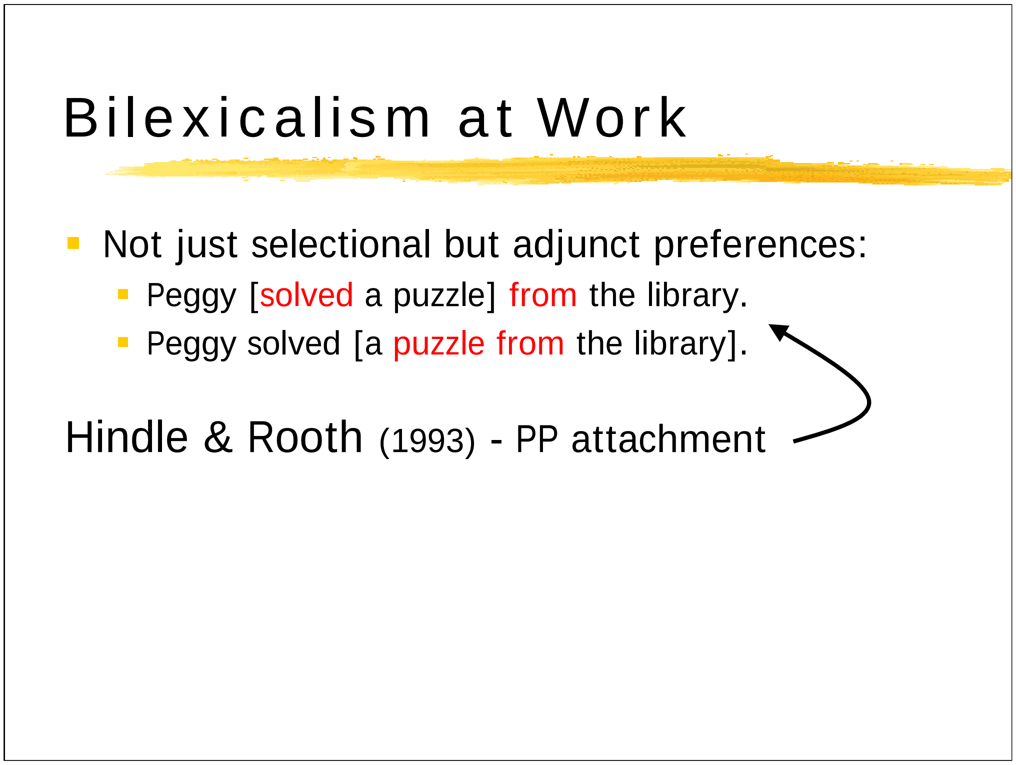
This is an algorithms paper, so here’s the formalism. Very easy:

We have a CFG in CNF.

All the rules look like these - i.e., a nonterminal constituent headed by word x must have a subconstituent also headed by x. Really just X-bar theory, head projection.

And typically, one gives these rules probabilities that depend on the words involved.

Such a grammar has <u>lots</u> of nonterminals - and that’s going to be a problem. Every nonterminal has a black part - a traditional nonterminal like NP, VP, S- plus a red part - a <u>literal</u> specification of the headword.



Bilexicalism at Work

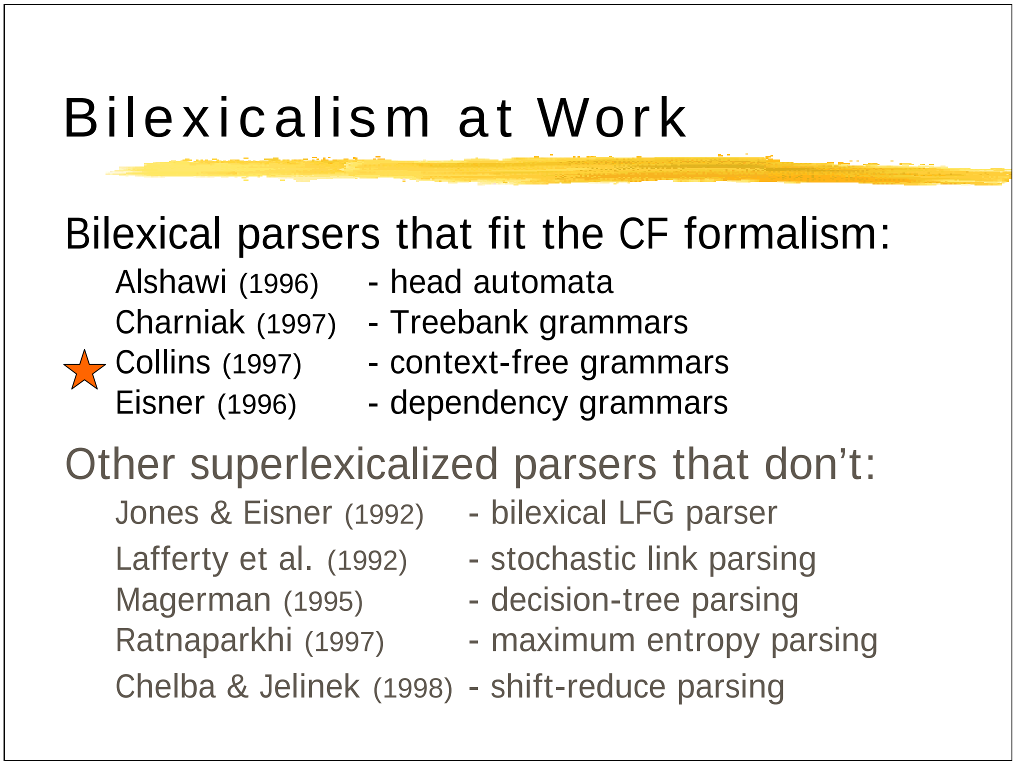
- Not just selectional but adjunct preferences:
  - Peggy [solved a puzzle] from the library.
  - Peggy solved [a puzzle from the library].

Hindle & Rooth (1993) - PP attachment

The rest of this talk will be about using such grammars efficiently.

Naturally, I want to claim that the work has some practical relevance, so I’ll point to past work.

First of all, notice that such grammars specify not just selectional but adjunct preferences. Which is more likely, *solve from* or *puzzle from*? Hindle and Rooth picked up on that idea in their unsupervised PP attachment work, which was one of the earliest - not quite the earliest - pieces of work to use bilexical statistics.



Bilexicalism at Work

Bilexical parsers that fit the CF formalism:

Alshawi (1996)	- head automata
Charniak (1997)	- Treebank grammars
★ Collins (1997)	- context-free grammars
Eisner (1996)	- dependency grammars

Other superlexicalized parsers that don't:

Jones & Eisner (1992)	- bilexical LFG parser
Lafferty et al. (1992)	- stochastic link parsing
Magerman (1995)	- decision-tree parsing
Ratnaparkhi (1997)	- maximum entropy parsing
Chelba & Jelinek (1998)	- shift-reduce parsing

More broadly, I'll point to the recent flurry of excellent bilexical parsers, and note that the ones here (for example) all fall within this formalism. Oh yes, they use different notation, and yes, they use different probability models, which is what makes them interesting and distinctive. But they're all special cases of the same simple idea, bilexical context-free grammars. So today's trick should in principle make them all faster.

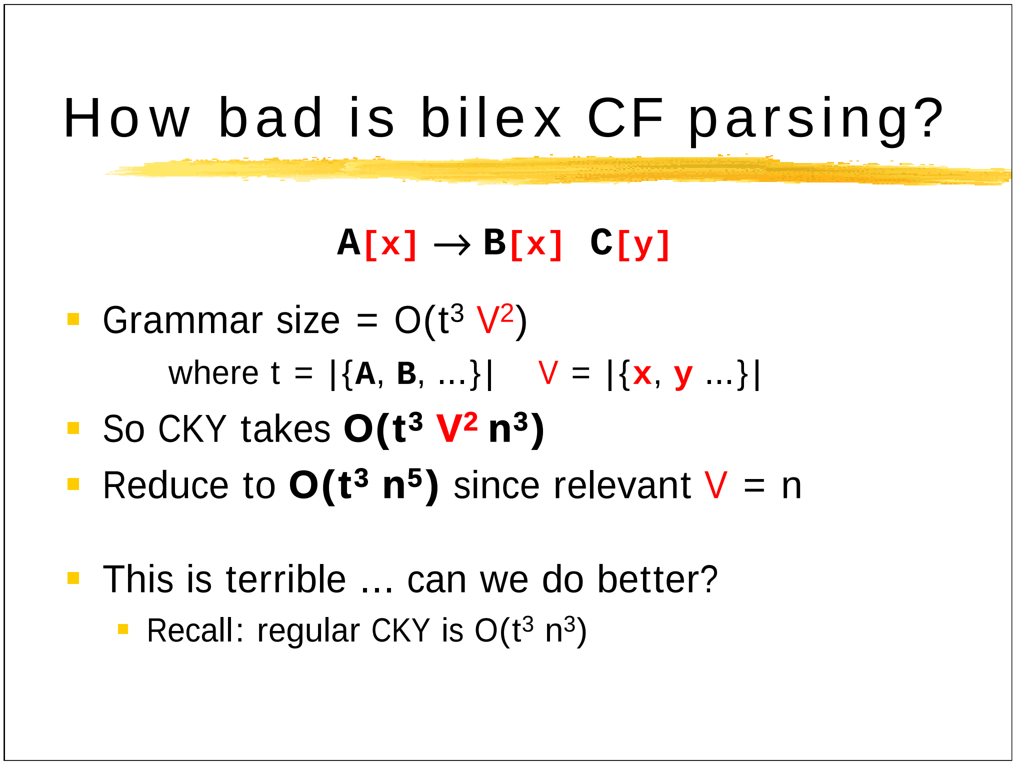
On the other hand, there are some other excellent superlexicalized parsers that are beyond the scope of this trick. Why don't they fit? Three reasons that I know of:

- Some of them are more than CF, like Jones & Eisner 1992, which was an LFG parser so it used unification.

- Some of them consider three words at a time, like Lafferty, Sleator and Temperley for link grammars - also very interesting, early, cubic-time work - or even four words at a time, like Collins and Brooks for PP attachment.

- And most of these probability models below aren't PCFGs. They're history based - they attach probabilities to moves of a parser. But the parsers above are declarative - they put probabilities on rules of the grammar - and they are bilexical PCFGs.

So we have all these parsers - how fast can they go?



How bad is bilex CF parsing?

$\mathbf{A[x] \rightarrow B[x]\ \ C[y]}$

- Grammar size = $O(t^3\ V^2)$

  where $t = |\{\mathbf{A}, \mathbf{B}, ...\}|$ $V = |\{\mathbf{x}, \mathbf{y}\ ...\}|$
- So CKY takes $\mathbf{O(t^3\ V^2\ n^3)}$
- Reduce to $\mathbf{O(t^3\ n^5)}$ since relevant $V = n$
- This is terrible ... can we do better?
  - Recall: regular CKY is $O(t^3\ n^3)$

We have to consider all rules of this form - for each such rule, we'll have to look it up to find out whether it's in the grammar and what its probability is.

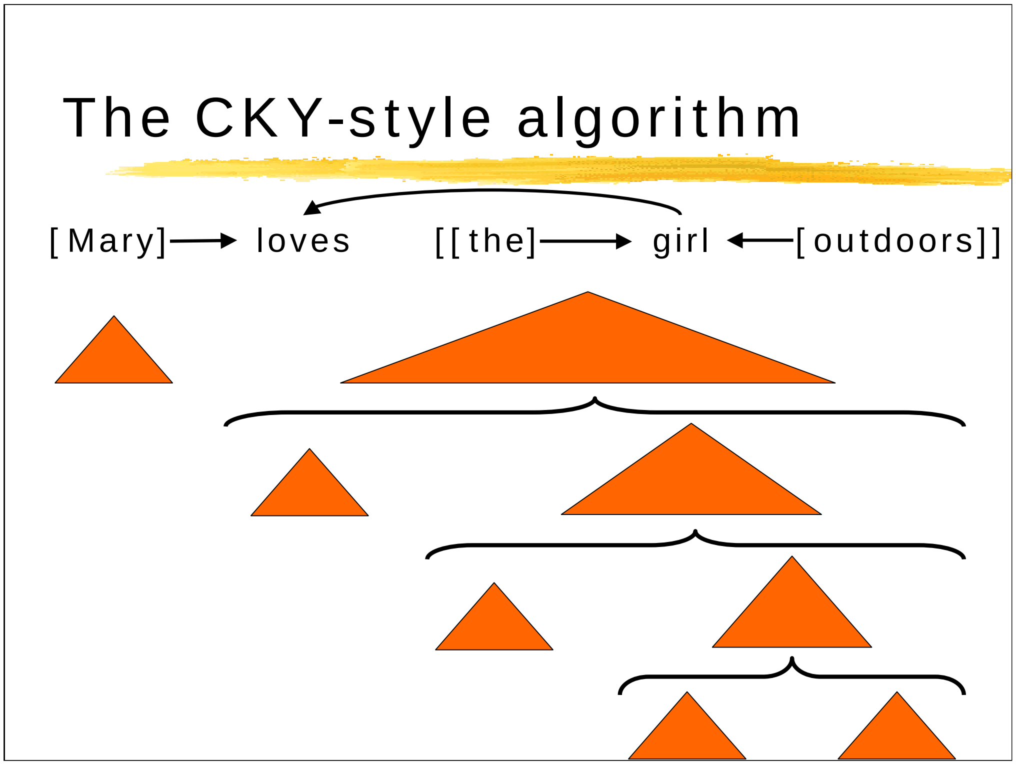
How many rules are there? Well, there are 3 black things (A,B,C) and 2 redthings (x,y). So the number of rules to consider is the number of black things cubed times the number of red things squared. The black things are just the traditional nonterminals, so there aren't too many for them; but V is the size of the vocabulary of (say) English, so V squared is really large.

Fortunately, if you're parsing a 30-word sentence, and you're concentrating really hard, then English vocabulary is just 30 words as far as you're concerned. You can ignore all other words while you're parsing this sentence. So we can replace V by the size of the relevant vocabulary, the number of distinct words in the sentence, which is at most n.

So in fact we get an n^5 algorithm. And in fact, Alshawi, Charniak, Collins, Carroll & Rooth all tell me that their parsers are asymptotically n^5.

But n^5 is bad. Heck, n^5 is terrible. The only reason we can parse at all with an n^5 algorithm is by pruning the chart heavily.

And remember, CKY is 2 factors of n better, one for each head. I'm going to show you how to get those factors of n back, one at a time.



The CKY-style algorithm

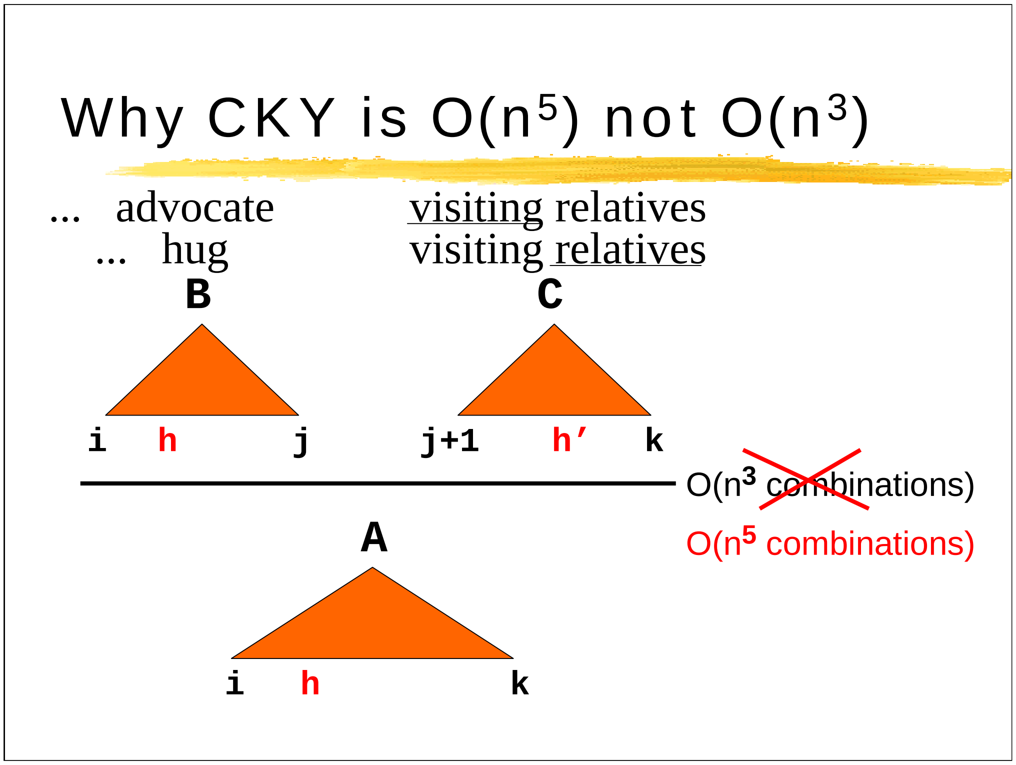
[Mary] → loves [[the] → girl ← [outdoors]]

Well, let’s see exactly what’s going wrong with CKY, and maybe we can fix it.

Triangles are subtrees, i.e., constituents.

Girl combines with the postmodifier “outdoors” to make an N-bar or something.

That combines with the determiner to its left to make an NP ...



Why CKY is O(n^5) not O(n^3)

... advocate visiting relatives

... hug visiting relatives

~~O(n^3 combinations)~~

O(n^5 combinations)

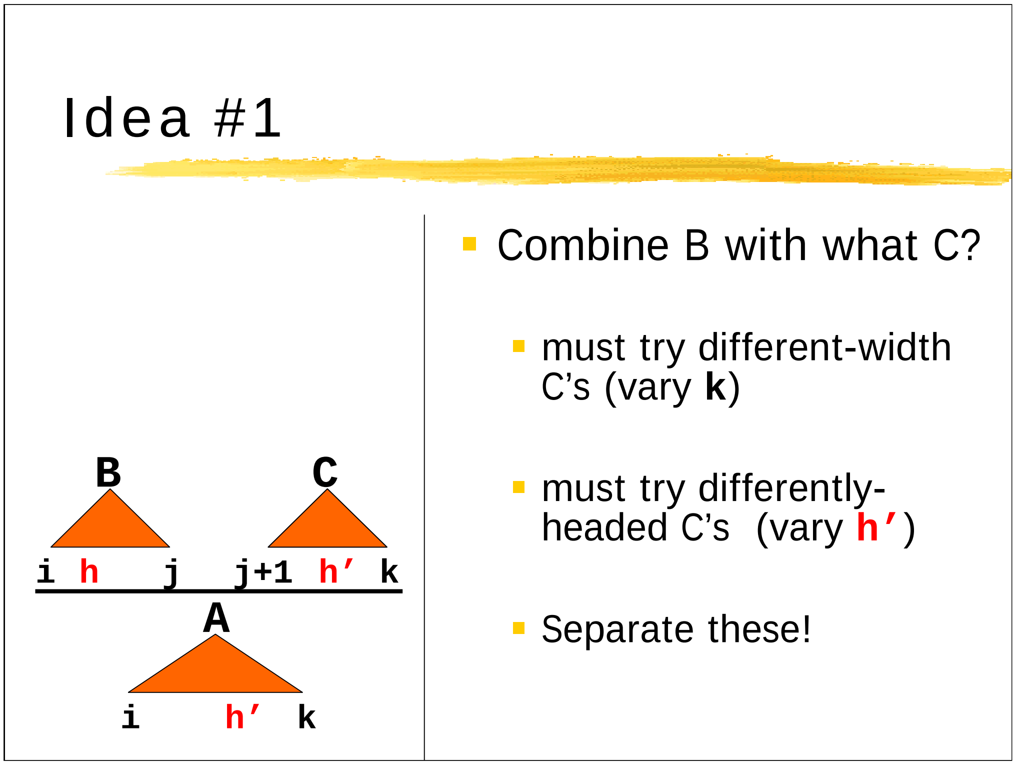
This is the derivation rule for CKY. If we have the two constituents above the line - a B right next to a C, notice that B ends at position j and C picks up at j+1 - then we can derive the constituent below the line, combining B and C to get A, if this is allowed by the grammar.

Here i, j, k can range from 1 to n, so there are n^3 ways of instantiating this rule: n^3 places we have to try it.

But if we're bilexical, these subtrees can combine only if their heads are compatible.

Well, now we have 5 numbers to keep track of - i, j, k, and the head positions h and h'. So the bilexical version of CKY is n^5.

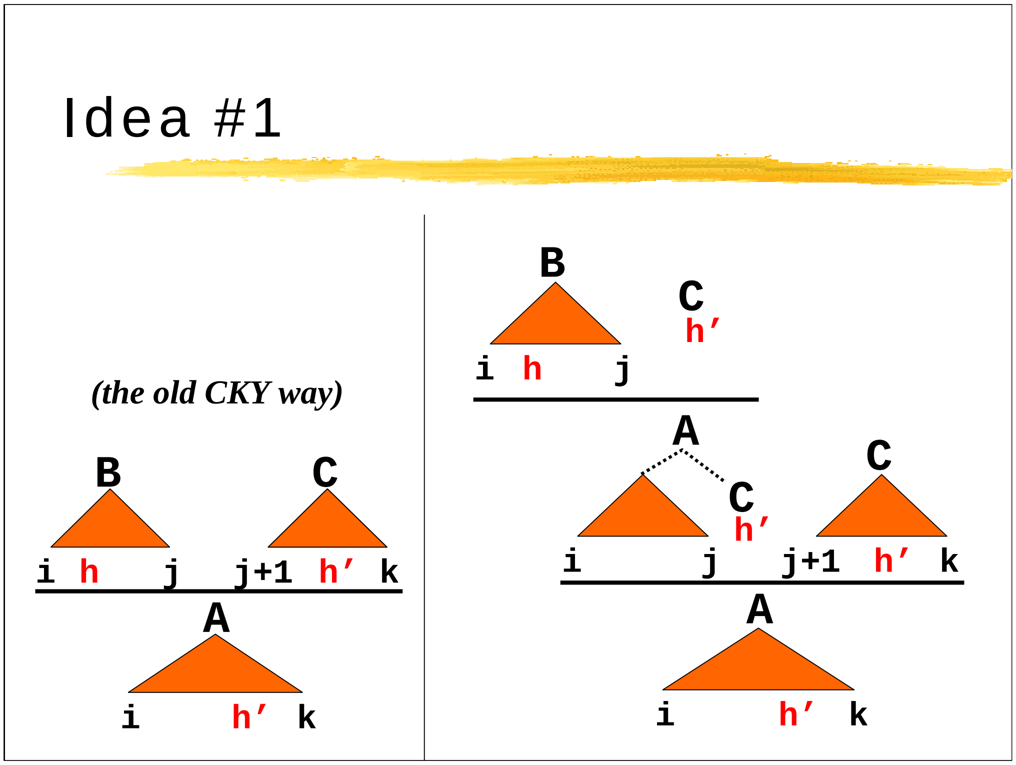
To make this concrete, think about Chomsky's example *visiting relatives* (also known as *flying planes*). This can be an NP headed by *visiting* -- visiting of relatives -- or an NP headed by *relatives* -- relatives who are visiting. That's two hypotheses of the same type, so the label C is NP in both cases, and they cover the same words, so j and k are also the same; they differ only in their head. And we have to keep both hypotheses, because perhaps one of them won't combine with the rest of the sentence. If the preceding word is *hug*, you need this hypothesis; while if the preceding word is *advocate*, you need this one.



Idea #1

- Combine B with what C?
  - must try different-width C's (vary **k**)
  - must try differently-headed C's (vary **h'**)
  - Separate these!

Well, here's an idea. When we're trying to combine B with some possible C -- *visiting relatives* and so on -- we have to try C's of different widths, that end at different places k. And we also have to try C's with different heads. Let's try fulfilling those two requirements separately. I'll call this a "two-step," in honor of the ACL swing banquet on Wednesday.



Idea #1

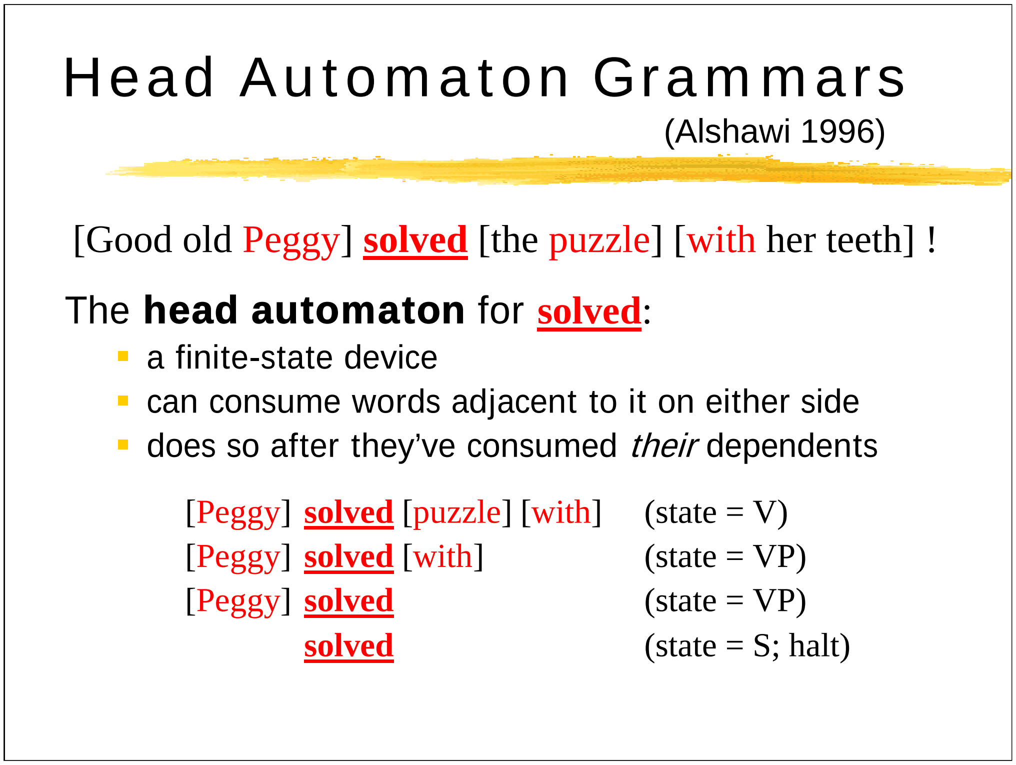
(the old CKY way)

Instead of combining tree B with all of tree C, let's just combine it with the disembodied head word h'. We can do that if h' (on the right) is a legal parent for h (on the left).

Ok, in comes the head, we combine it, and the result is this funny structure. Roughly speaking, this is what categorial grammar would call an A/C. It's an A missing its head child on the right, a C headed at position h'. And if such a C comes along, we can attach it to get our A as before.

So that's the two step - first attach the head, then the full constituent.

Notice that in the first step, we don't know yet where this C tree might end - we only know h', not k. So we're only considering 4 numbers -- n^4. And the result of this first step doesn't have to mention h, because we won't need it anymore - we've already checked it against h'. So the second step doesn't mention h, and it too only considers 4 numbers. So the whole algorithm is n^4.



Head Automaton Grammars

(Alshawi 1996)

[Good old Peggy] **solved** [the puzzle] [with her teeth] !

The **head automaton** for **solved**:

- a finite-state device
- can consume words adjacent to it on either side
- does so after they've consumed *their* dependents

[Peggy] **solved** [puzzle] [with]	(state = V)
[Peggy] **solved** [with]	(state = VP)
[Peggy] **solved**	(state = VP)
solved	(state = S; halt)

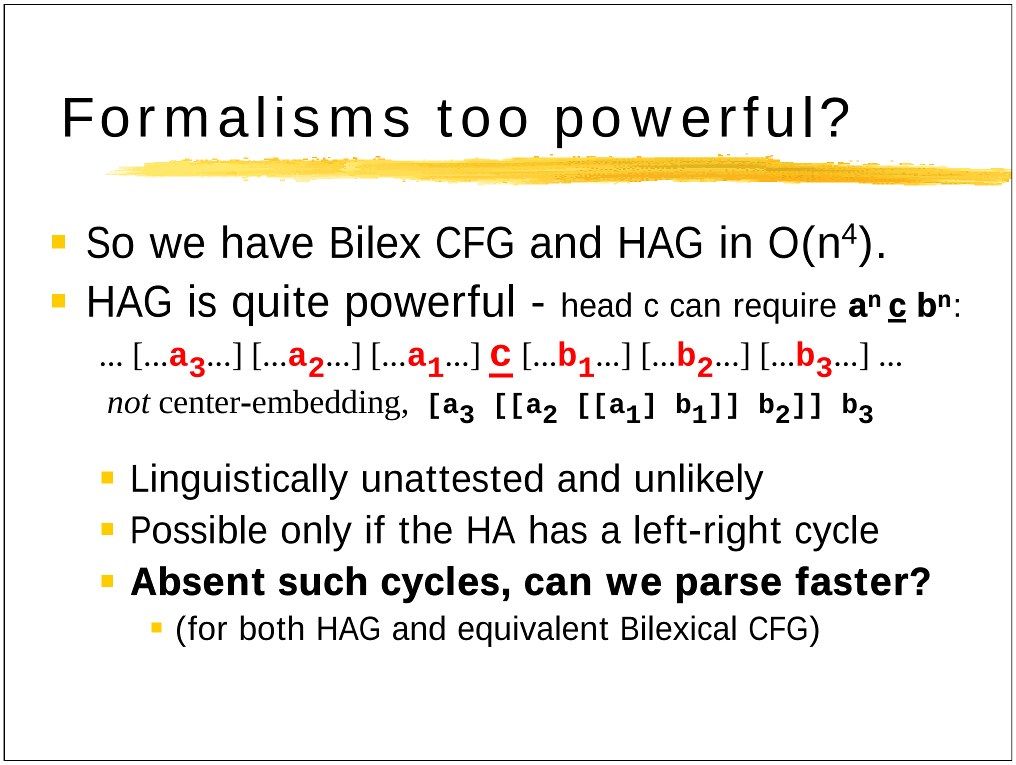
Now I'm going to change gears a little. I want to make a point about bilexical grammars that is easier to make in an equivalent formalism.

If you were at ACL-96, you may remember Hiyan Alshawi's talk on Head Automaton Grammars. Suppose we want to judge the grammaticality of this sentence, using a grammar that associates a so-called head automaton with every word in the language. The automaton checks the dependents of the word.

A head automaton is a finite-state device -- it's read-only -- but instead of just reading one string of input, it sits between two input strings and it can consume dependent words on either side. And this is recursive.

So here, these words in brackets have already checked and consumed their dependents. The automaton for *solved* now asks: is this an okay sequence of dependents for *solved*? *Peggy* on the left, *puzzle with* on the right?

Well, *solved* is a verb, so it starts in state V. By eating its object, *puzzle,* it becomes a verb phrase, and transitions to state VP; it can do that because *puzzle* is an okay object for *solved*. From VP it can eat a prepositional-phrase adjunct and loop back to VP. Now it can take its subject, and transition to state S. And since S is a halt state, we conclude that this dependent sequence was okay, that what we started with was a grammatical phrase headed by *solved*.



Formalisms too powerful?

- So we have Bilex CFG and HAG in O(n^4).
- HAG is quite powerful - head c can require $\mathbf{a}^n\ \underline{\mathbf{c}}\ \mathbf{b}^n$:

  ... [...a_3...] [...a_2...] [...a_1...] $\underline{c}$ [...b_1...] [...b_2...] [...b_3...] ...

  not center-embedding, $[a_3\ [[a_2\ [[a_1]\ b_1]]\ b_2]]\ b_3$

  - Linguistically unattested and unlikely
  - Possible only if the HA has a left-right cycle
  - **Absent such cycles, can we parse faster?**
    - (for both HAG and equivalent Bilexical CFG)

Head automaton grammars are obviously bilexical, and they're basically equivalent to bilexical context-free grammars; that's easy to show. So the trick I already showed you applies, and we can parse with them in time n^4 rather than n^5.

Can we do better?

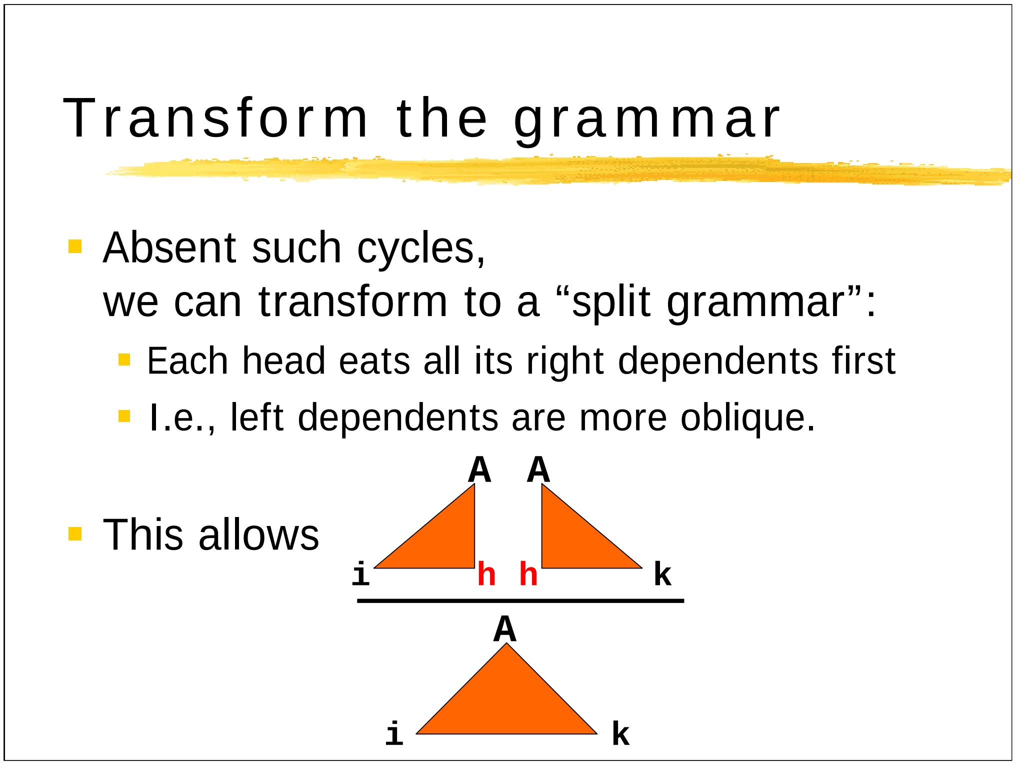
Well, I want to pick up on a comment of Alshawi's that HAGs are quite powerful. We can write a head automaton for a word *c* that requires *c* to take the same number of dependents to the left and to the right. The automaton has a cycle -- it says, consume *a* to the left, then *b* to the right, then again *a* to the left, then *b* to the right, then again and so on.

This is not center-embedding. In a center-embedded sentence like *The cat the rat the policeman beat chased ran*, each word has one dependent, and the dependencies are nested. Here, the single word *c* takes 2n dependents, n on each side.

Well, I don't know any head in any natural language that behaves that way.

We can get this kind of funny behavior only if the HA contains a cycle on which it reads from both the left and the right.

Suppose our head automaton grammar has no such cycles - since I know of no linguistic reason for them. Or our bilexical CFG satisfies the equivalent condition. Can we then parse faster?

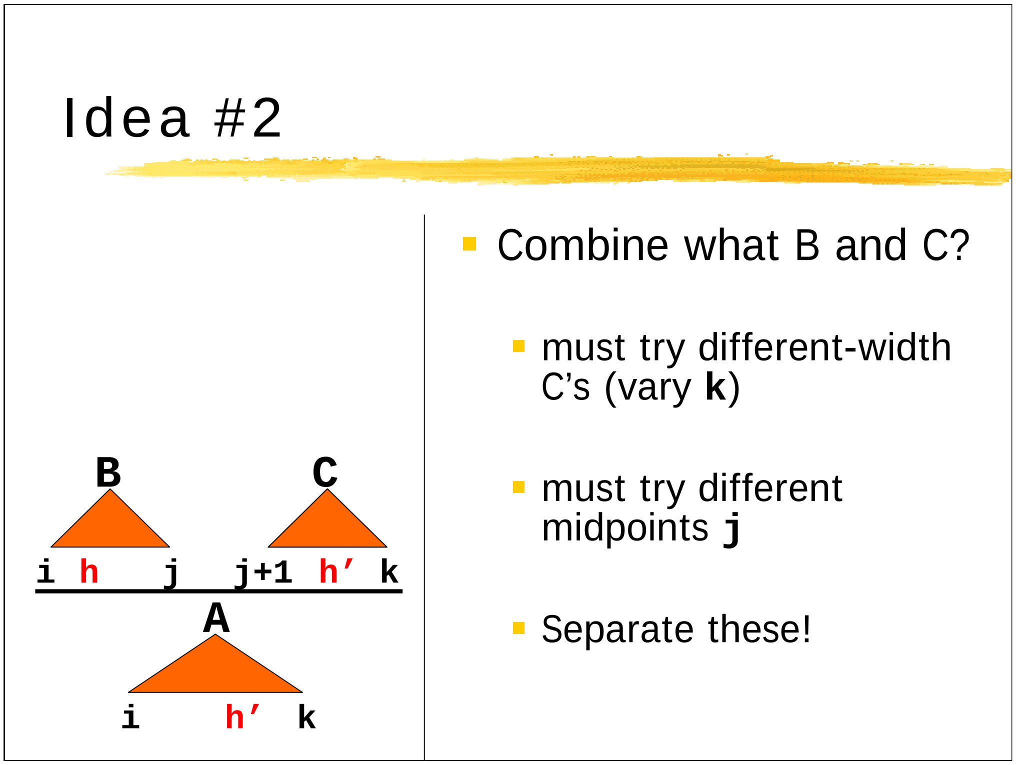


Transform the grammar

- Absent such cycles,
  we can transform to a “split grammar”:
  - Each head eats all its right dependents first
  - I.e., left dependents are more oblique.
- This allows

If we don’t have such cycles, we can transform to what I call a “split grammar” in which each head eats its right dependents and then its left dependents.

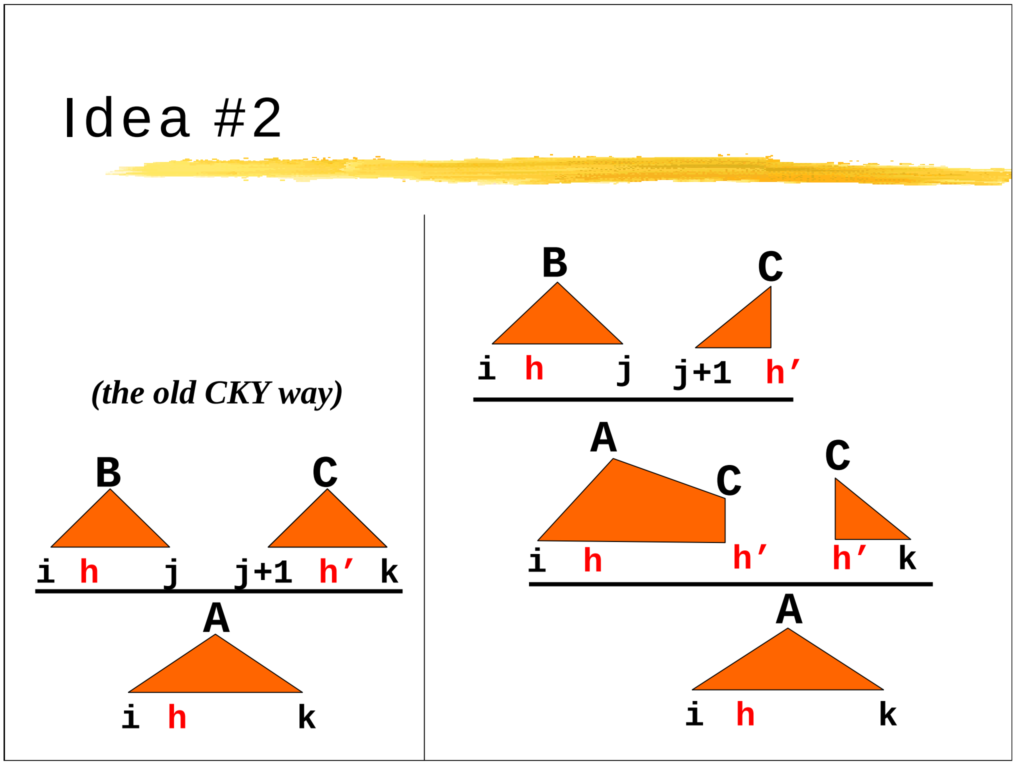
Then we have this interesting new option - head h can acquire its left children separately from its right children, and we can stick those two half trees together later on. Maybe it’s not quite true that any sequence of left children is compatible with any sequence of right children -- but the amount of information the two half-trees need in order to check compatibility is finite and bounded, and can be stored at h.



Idea #2

- Combine what B and C?
  - must try different-width C’s (vary **k**)
  - must try different midpoints **j**
  - Separate these!

So again, well try a 2-step. '



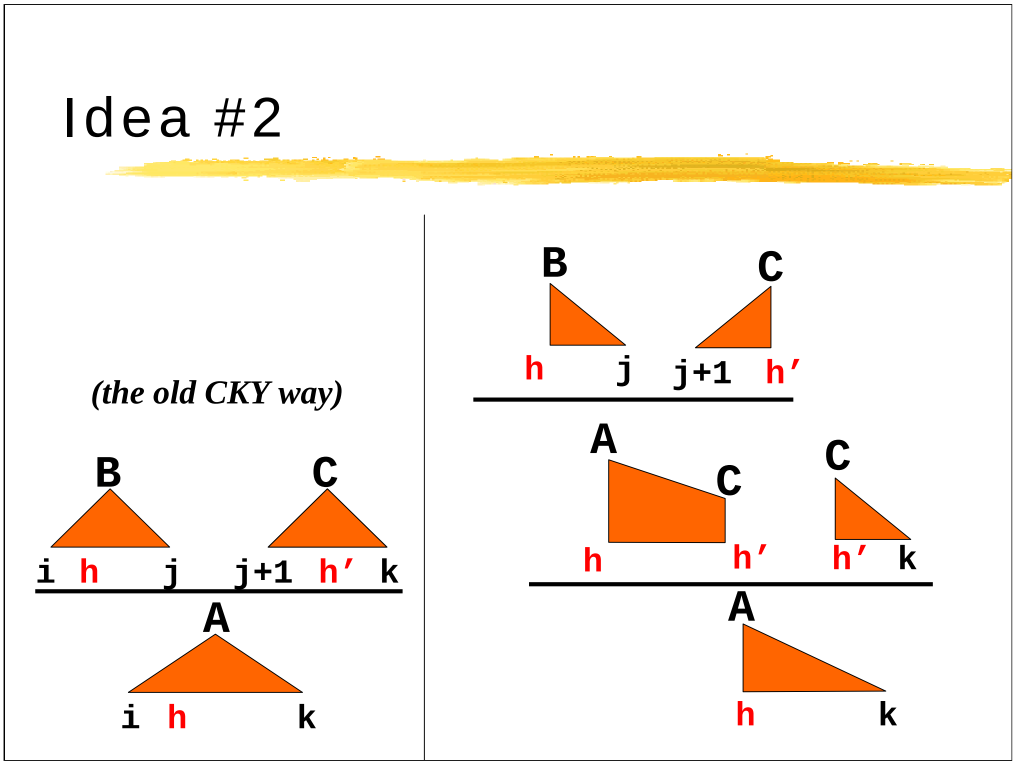
Idea #2

(the old CKY way)

We combine B with just the left half of C: the triangle represents head h’ together with just its left children. And we get this funny object - a new polygon - which we combine with the right half of C.

Again, this is an n^4 algorithm. It’s a slightly different n^4 algorithm from before: instead of forgetting h halfway through, at the new polygon, we forget the midpoint j. We’re only sticking on the right half of C later, and it doesn’t matter where the left half starts - the two halves are independent.

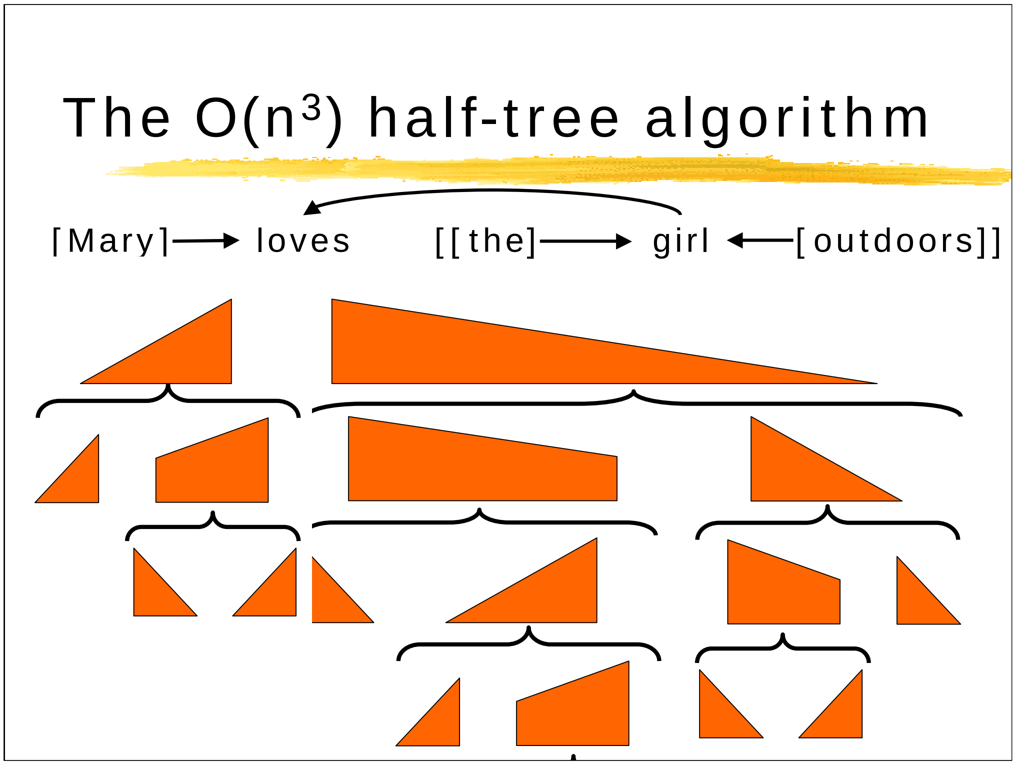
So this is n^4 - now let’s make it n^3. If we can wait to add k at the right ...



Idea #2

(the old CKY way)

... why not wait to add i at the left? Now we’re only considering 3 positions per step, so we’re n^3.



The O(n^3) half-tree algorithm

[Mary] → loves [[the] → girl ← [outdoors]]

That’s the idea, and rather than take you through the formal derivation rules, which are in the paper, I’ll just animate the new n^3 algorithm.

We seed the chart like this. Each word - like *girl* - has a 0-width left triangle indicating it has no left kids yet, and a 0-width right triangle indicating it has no right kids yet.

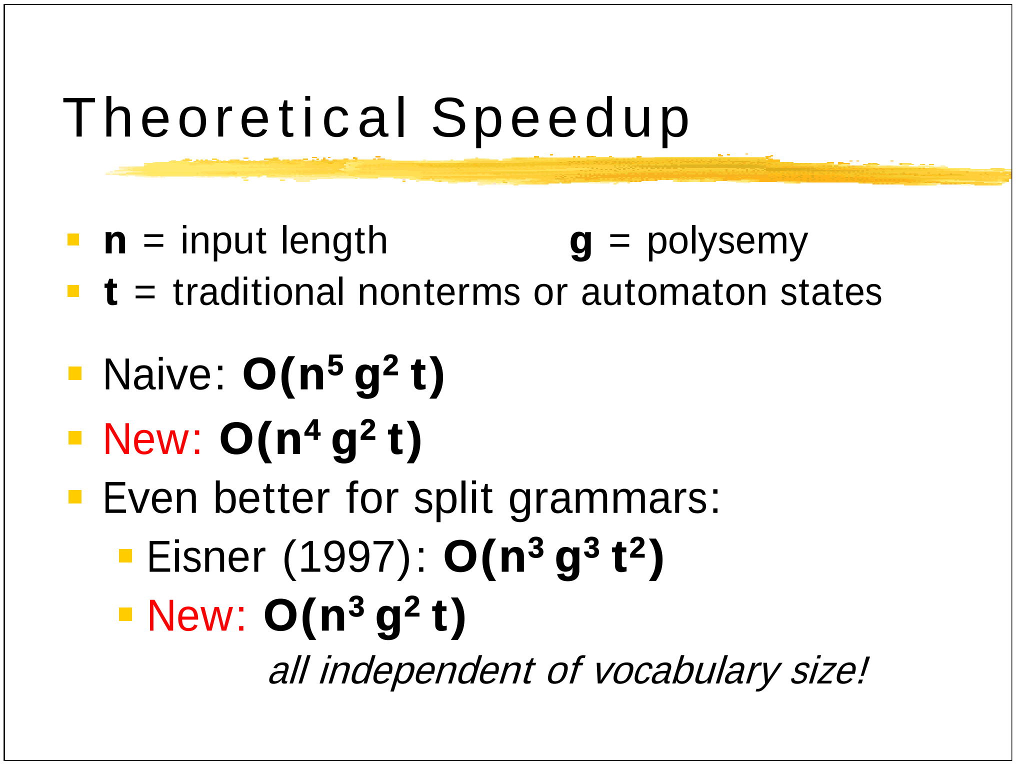
girl takes the left half of its dependent *outdoors*, then separately the right half of *outdoors*. That’s the two-step we just saw. So *girl* has acquired a right child.

Now the left half of *girl* gets its left kids similarly, another two-step.

So now we have *girl* with its left dependents, and *girl* with its right dependents. Do we put those together? Nope - then the head would be buried in the middle, and we’d be back up to n^4. So we don’t put them together yet, we do the 2-step all over again. *loves* takes the left half of the NP, then the right half.

Similarly, *loves* gets its left child. And now we’re back again to the two halves of the sentence. If this sentence is an argument to a word on the left -- *I believe Mary loves the girl outdoors* -- then that word, *believe*, will suck up these two halves one at a time ...

Wanna see it again? ...



Theoretical Speedup

- **n** = input length **g** = polysemy
- **t** = traditional nonterms or automaton states

- Naive: $\mathbf{O(n^5 g^2 t)}$
- New: $\mathbf{O(n^4 g^2 t)}$
- Even better for split grammars:
  - Eisner (1997): $\mathbf{O(n^3 g^3 t^2)}$
  - New: $\mathbf{O(n^3 g^2 t)}$

all independent of vocabulary size!

So here's a summary of the theoretical speedup.

Suppose n is the number of words in the sentence. t is the number of traditional nonterminals - NP, VP - or the maximum number of states per head automaton.

g is a grammar constant that can be regarded as a polysemy factor; for standard head automaton grammars without polysemy, g=1, and for bilexical CFG, g=t.

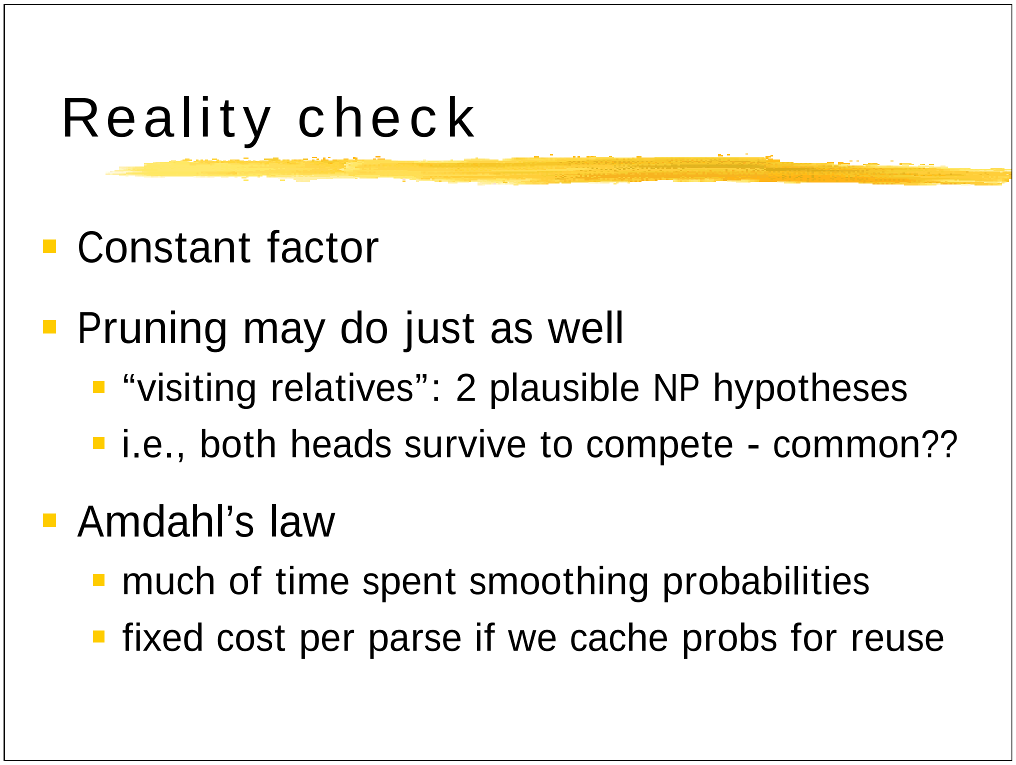
This is the naive CKY-style algorithm, n^5.

The first algorithm I showed you gets down to n^4.

For split grammars we can do even better. A couple of years ago at IWPT, I showed how to get down to n^3, but with a higher grammar constant. Our new n^3 algorithm, the one I just animated, gets back the old grammar constant.

Note: The g's and t's are all very confusing.

Here I am assuming a HAG framework where the automata are deterministic (otherwise it costs about a factor of t) and where only the word and not its nonterminal (= final state) is readable by its parent. If we want more to be readable, we should put g = t. Indeed, for standard bilex grammar, try putting g = t = number of nonterms. But in general, we can get away with g < t.



Reality check

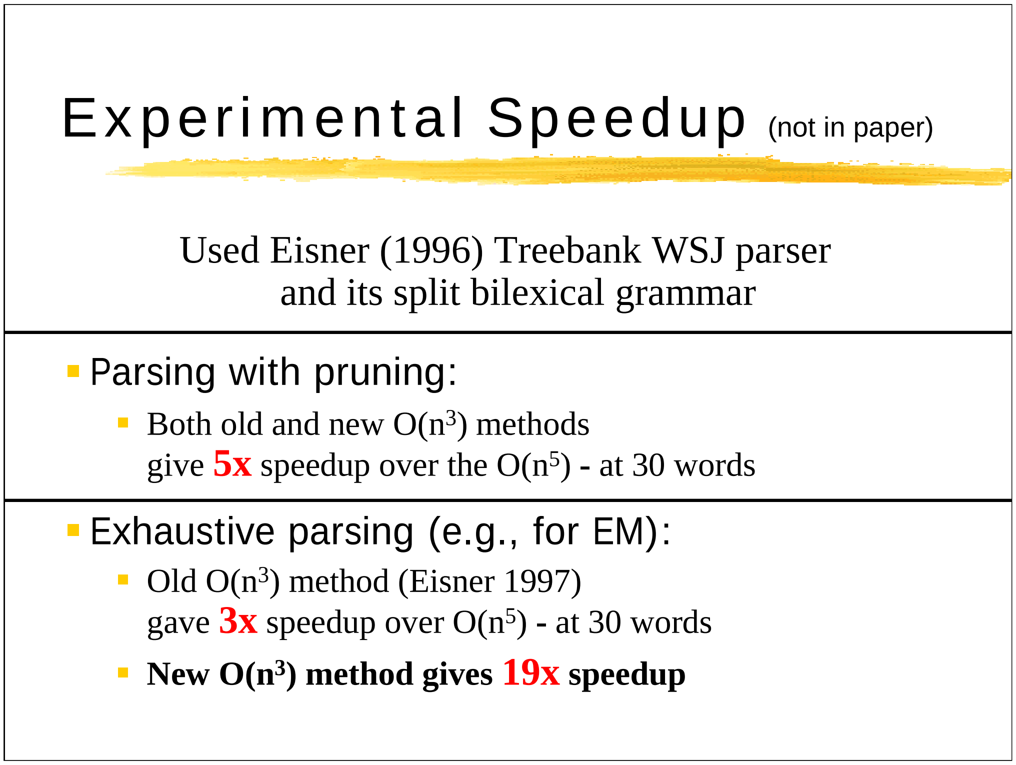
- Constant factor
- Pruning may do just as well
  - “visiting relatives”: 2 plausible NP hypotheses
  - i.e., both heads survive to compete - common??
- Amdahl’s law
  - much of time spent smoothing probabilities
  - fixed cost per parse if we cache probs for reuse

But that’s just an asymptotic speedup.

You might ask: does it translate into a real speedup for typical-length sentences, or does the constant factor kill us?

And there are other reasons to worry that this algorithm might not help much. Remember “visiting relatives” -- two NP hypotheses covering the same string, but with different heads. I used this example because you can see the ambiguity easily - i.e., both hypotheses are plausible and should survive pruning. But how often does that happen? It’s really hard to find such examples in English. In Chinese, maybe it happens more -- there’s lots of word segmentation ambiguity and part of speech ambiguity, and the chart tends to saturate rapidly. But in English at least, maybe a hard grammar or probabilitistic pruning will kill off most of the competitors.

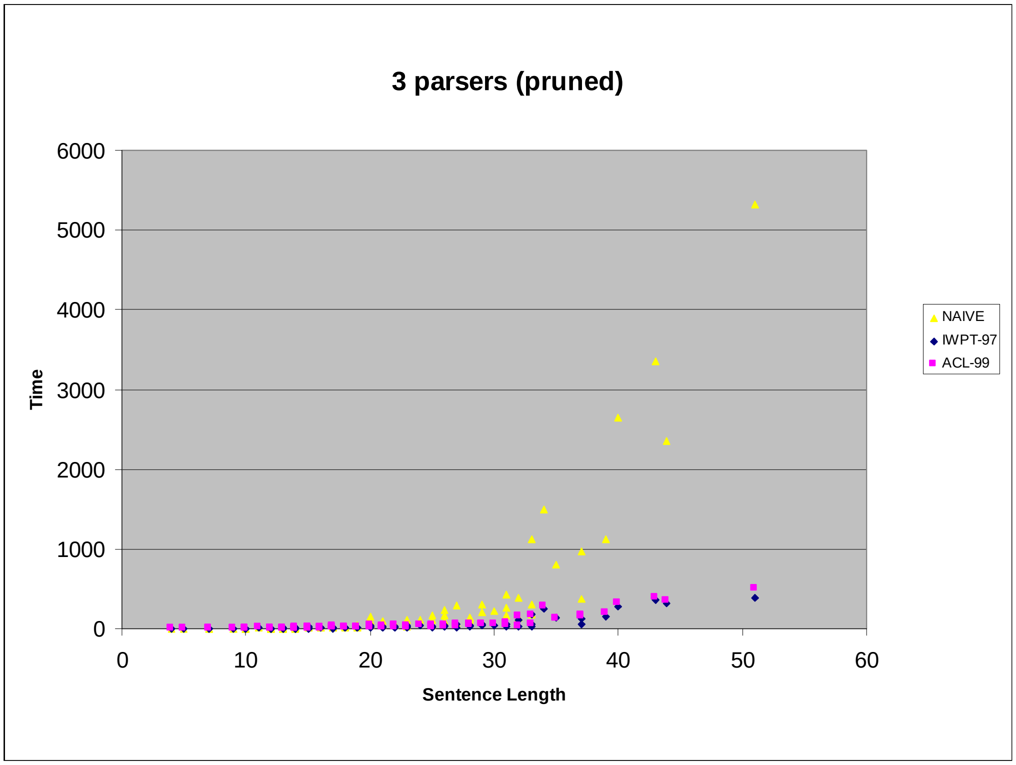
Finally, there’s Amdahl’s Law, which says that additive constant factors are important too. You can only get so far by speeding up a module that doesn’t account for much of the runtime. My own parser spends a lot of time elsewhere, smoothing probabilities. An n^5 algorithm and an n^3 algorithm use the same probabilities, it’s just that the n^5 algorithm refers to them more often - but the n^5 algorithm doesn’t have to waste any extra time recomputing them, since it cached them the first time. So n^5 versus n^3 doesn’t affect the part of the runtime that just computes probabilities. If that’s where we’re spending our time, the new algorithm won’t help so much.



Experimental Speedup (not in paper)

Used Eisner (1996) Treebank WSJ parser
and its split bilexical grammar

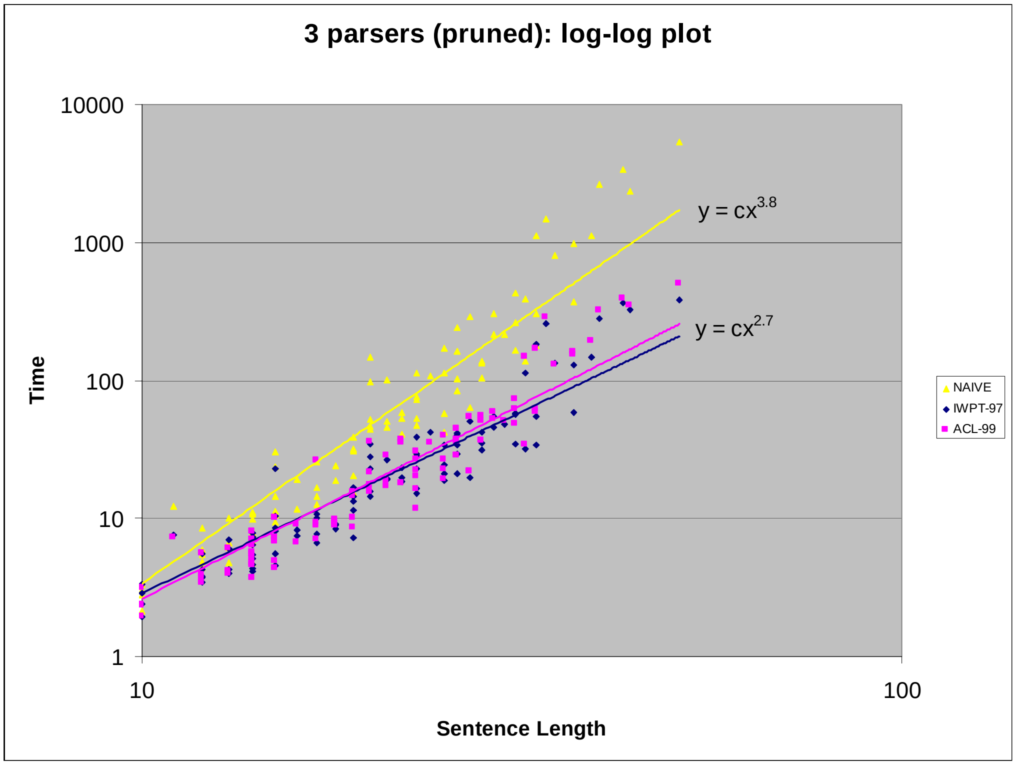
- Parsing with pruning:
  - Both old and new O(n^3) methods give **5x** speedup over the O(n^5) - at 30 words
- Exhaustive parsing (e.g., for EM):
  - Old O(n^3) method (Eisner 1997) gave **3x** speedup over O(n^5) - at 30 words
  - **New O(n^3) method gives 19x speedup**

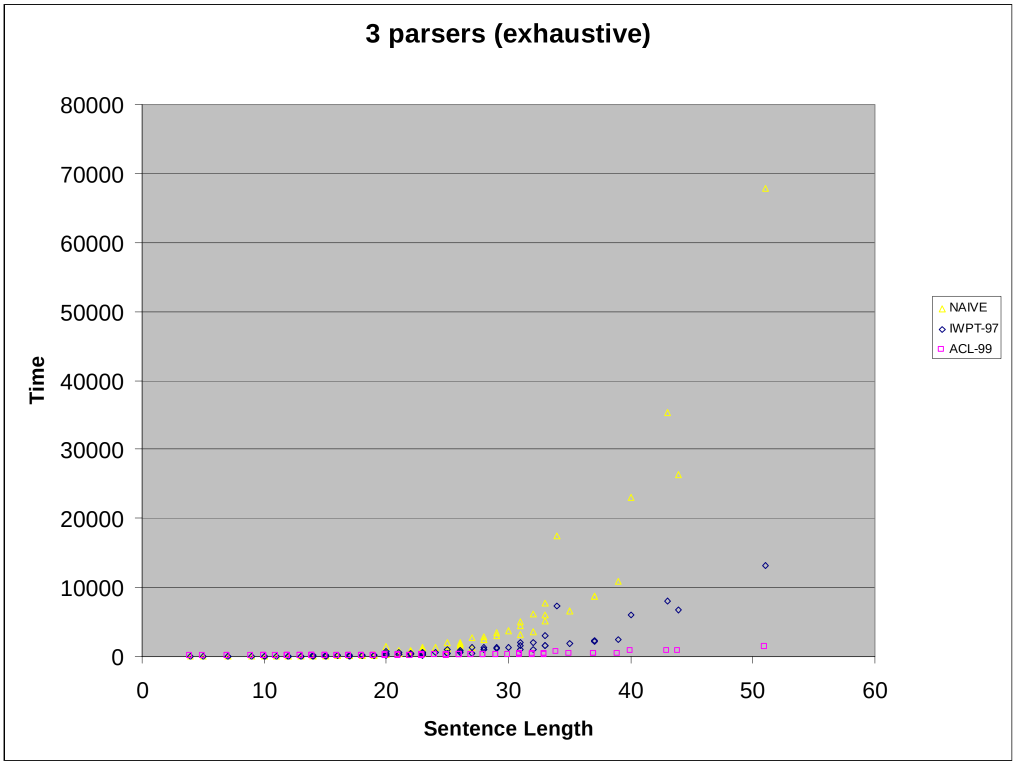


Performance.

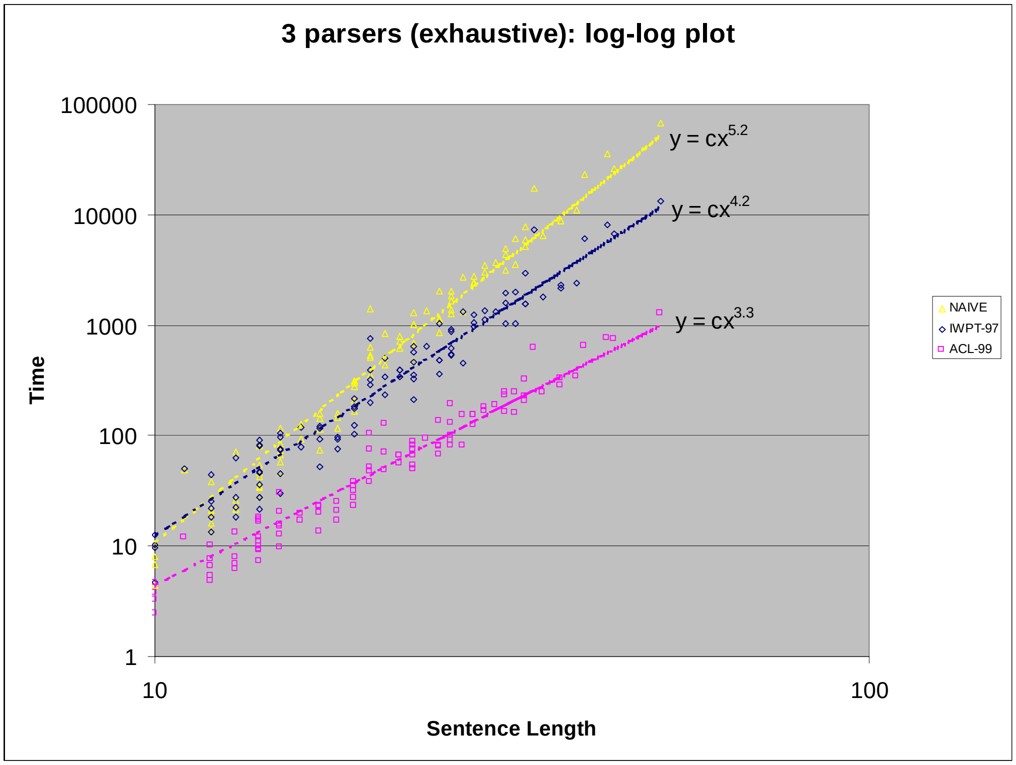
You can see that both the cubic-time algorithms - the one I reported in 1997, and the new one - are much better than CKY.

Actually, that’s a little hard to see for shorter sentences, so let’s try a log-log plot to spread the points apart a bit:

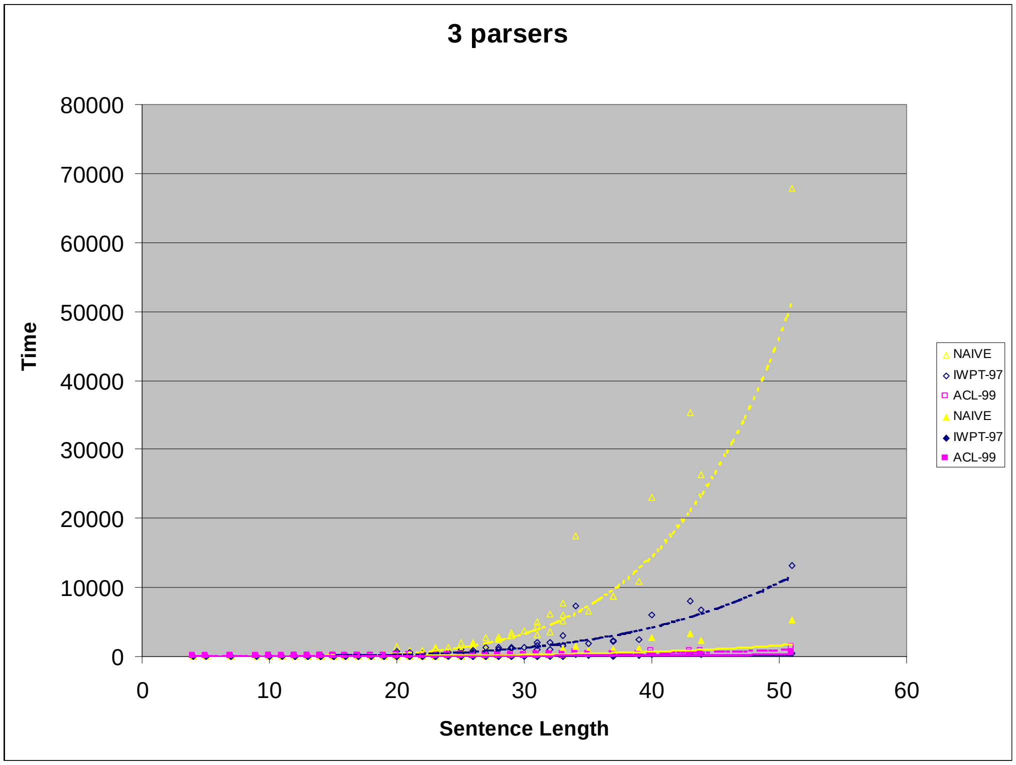

3 parsers (pruned): log-log plot
10000
1000
100
10
1
Time
10
100
Sentence Length
y = cx^3.8
y = cx^2.7
NAIVE
IWPT-97
ACL-99



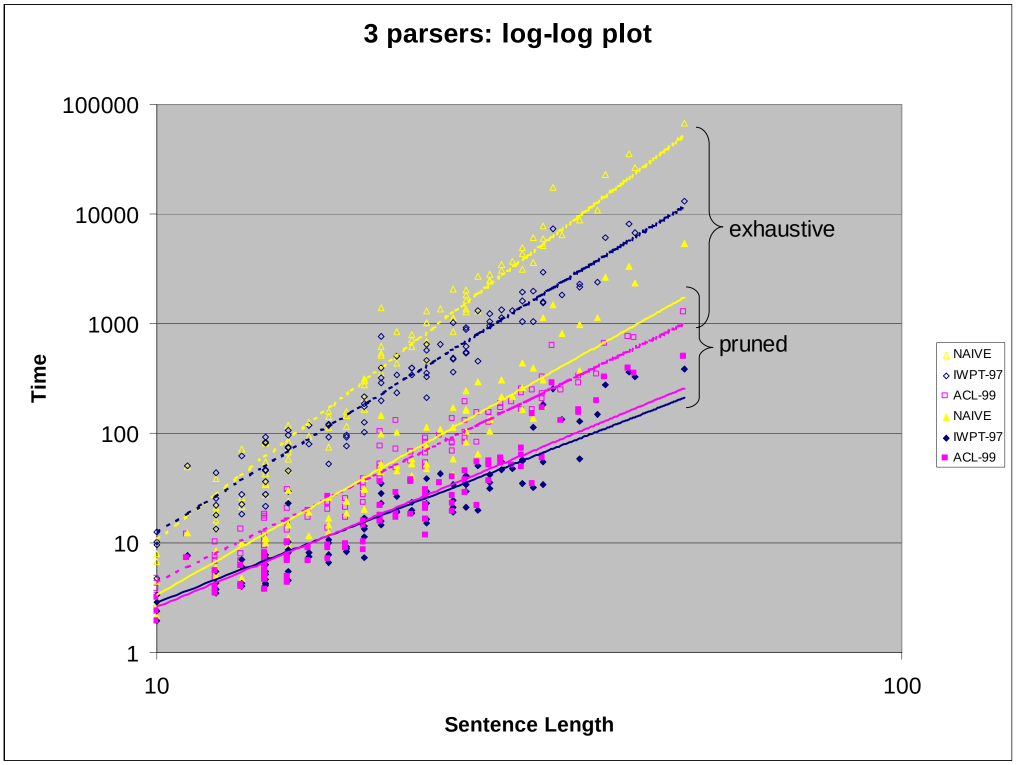
Now, sometimes you have to do exhaustive parsing - you're not allowed to prune. You might actually care about low-probability constituents. For example, it's important to keep them when running the Inside-Outside algorithm, so that they can grow in strength. Here's what things look like if we don't prune - now the new n^3 algorithm clearly beats the old n^3 as well as the naive n^5.



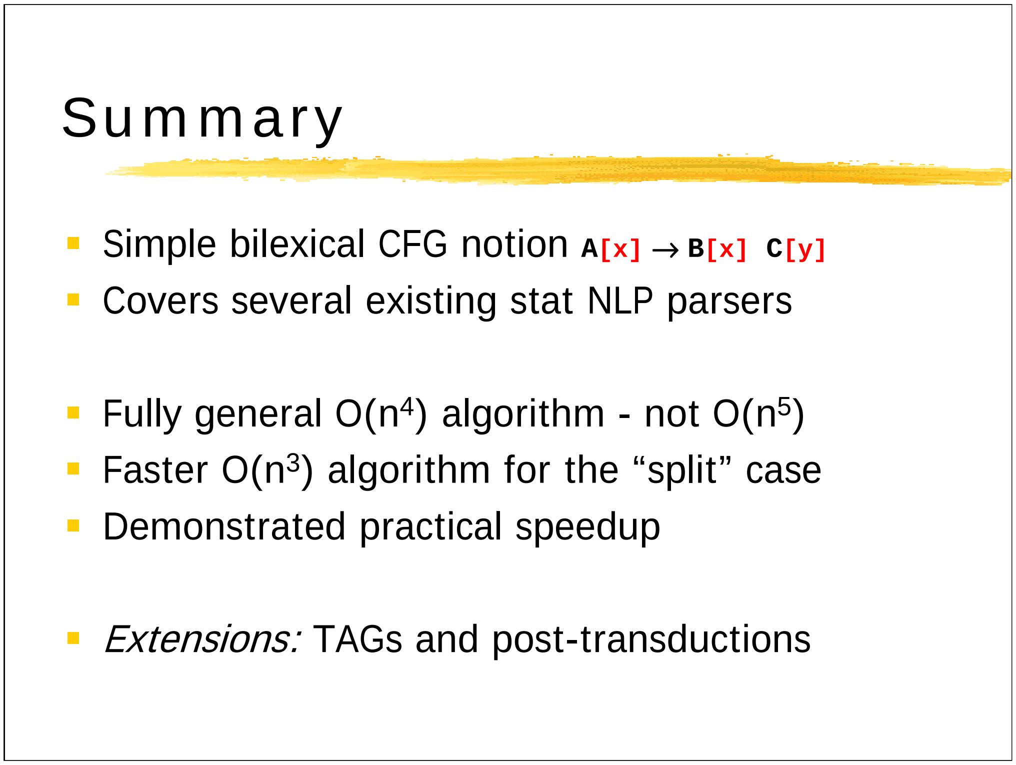
Again, here's a log-log plot, showing that the new algorithm wins at every sentence length. We expect the two n^3 lines (blue and pink) to be about parallel, since they differ only by a grammar factor that's independent of sentence length.



And, just for completeness, here are all the results together -- the hollow symbols are the exhaustive parsers, and of course that they take much longer than the pruned ones.



However, note that the new algorithm runs as fast without pruning as the naive algorithm runs with pruning!



Summary

- Simple bilexical CFG notion $A[x] \rightarrow B[x]\ C[y]$
- Covers several existing stat NLP parsers

- Fully general O(n^4) algorithm - not O(n^5)
- Faster O(n^3) algorithm for the “split” case
- Demonstrated practical speedup

- *Extensions:* TAGs and post-transductions

In summary:

Giorgio and I have given a simple formal notion of bilexical context-free grammar, which covers a lot of current work, for example head automaton grammars and a number of robust parsers out there.

We’ve shown how to parse in time n^4 rather than n^5, and we’ve identified a class of “split grammars” that is probably enough for linguistic purposesand can be parsed in time n^3.

Finally, we’ve shown that the algorithm really does speed things up in practice.

We also happen to have some extensions. Bilexicalized tree-adjoining grammars take time n^8 with the usual TAG algorithm; we know how to get this down to n^7. We’re also interested in the case of tree-insertion grammars. Another case we can handle efficiently in n^3 time is where the language is generated by a bilexical CFG composed with a transduction. Such a transduction is very useful for handling various messy stuff like polysemy, epsilon deletion, affix hopping, noisy input, and so on.