


6.863J Natural Language Processing

Lecture 7: parsing with hierarchical structures – context-free parsing



Robert C. Berwick
berwick@ai.mit.edu

The Menu Bar

- **Administrivia:**
 - Schedule alert: Lab2 due today; Lab 3 out late Weds. (context-free parsing)
 - Lab time today, tomorrow
 - Please read notes3.pdf, englishgrammar.pdf (on web)
- *Agenda:*
 - Chart parsing summary; time complexity
 - How to write grammars

Chart parsing summary



- Data structures & Algorithm
- Data structures
- A chart parser has three data structures:
 - an input stack, which holds the words of the input sentence (in order)
 - a chart, which holds completed phrases organized by starting position and length
 - a set of edges, organized by ending position.

Input sentence stack



- The input
- Positions in the input sentence will be numbered starting with zero and will be the positions between successive words. For example:

0 I 1 shot 2 an 3 elephant 4 in 5 my 6 pajamas 7

For now, assume POS already assigned,
words consumed l-to-r



We have presented the chart
graphically so far...

Chart

- Example of chart

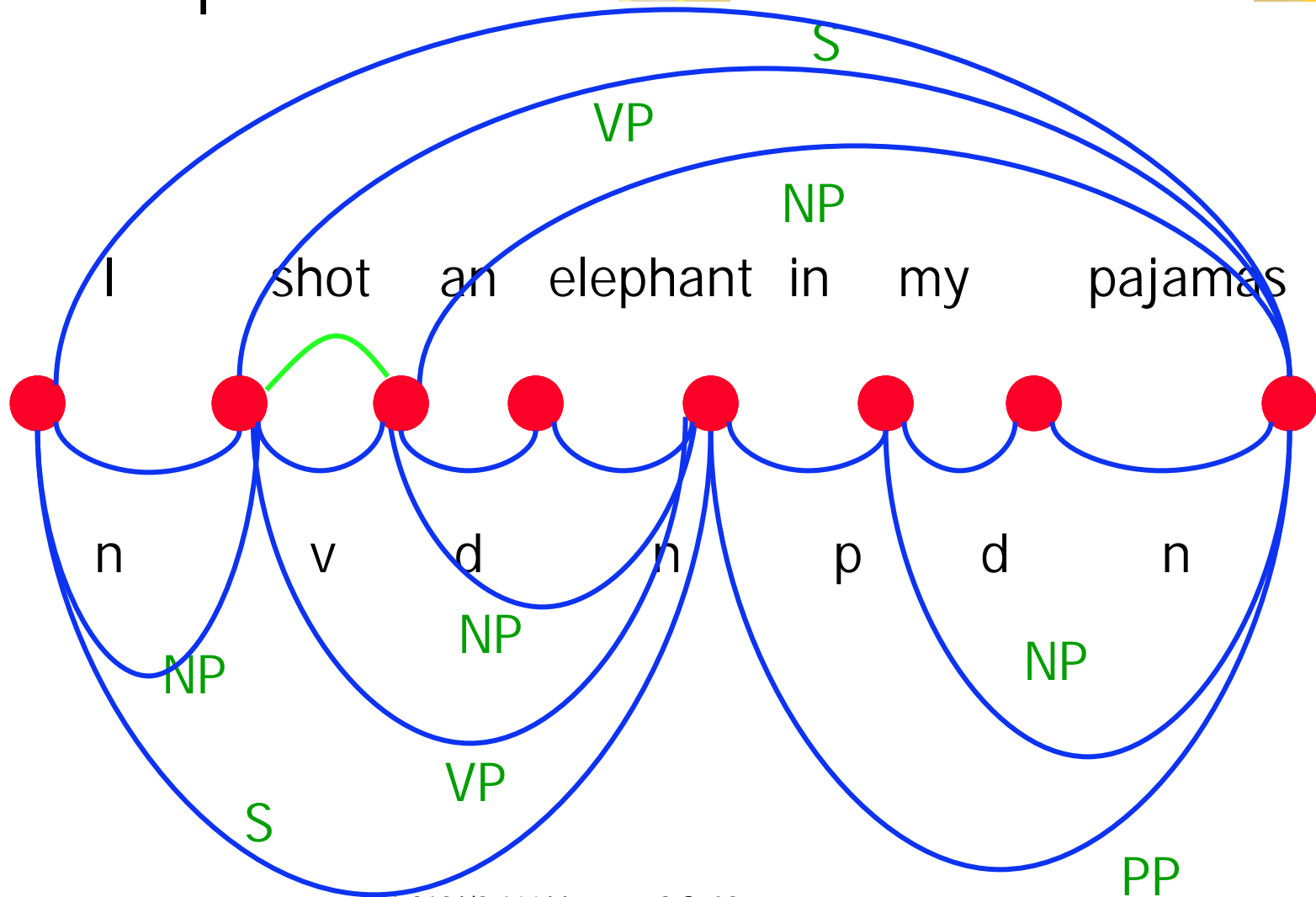


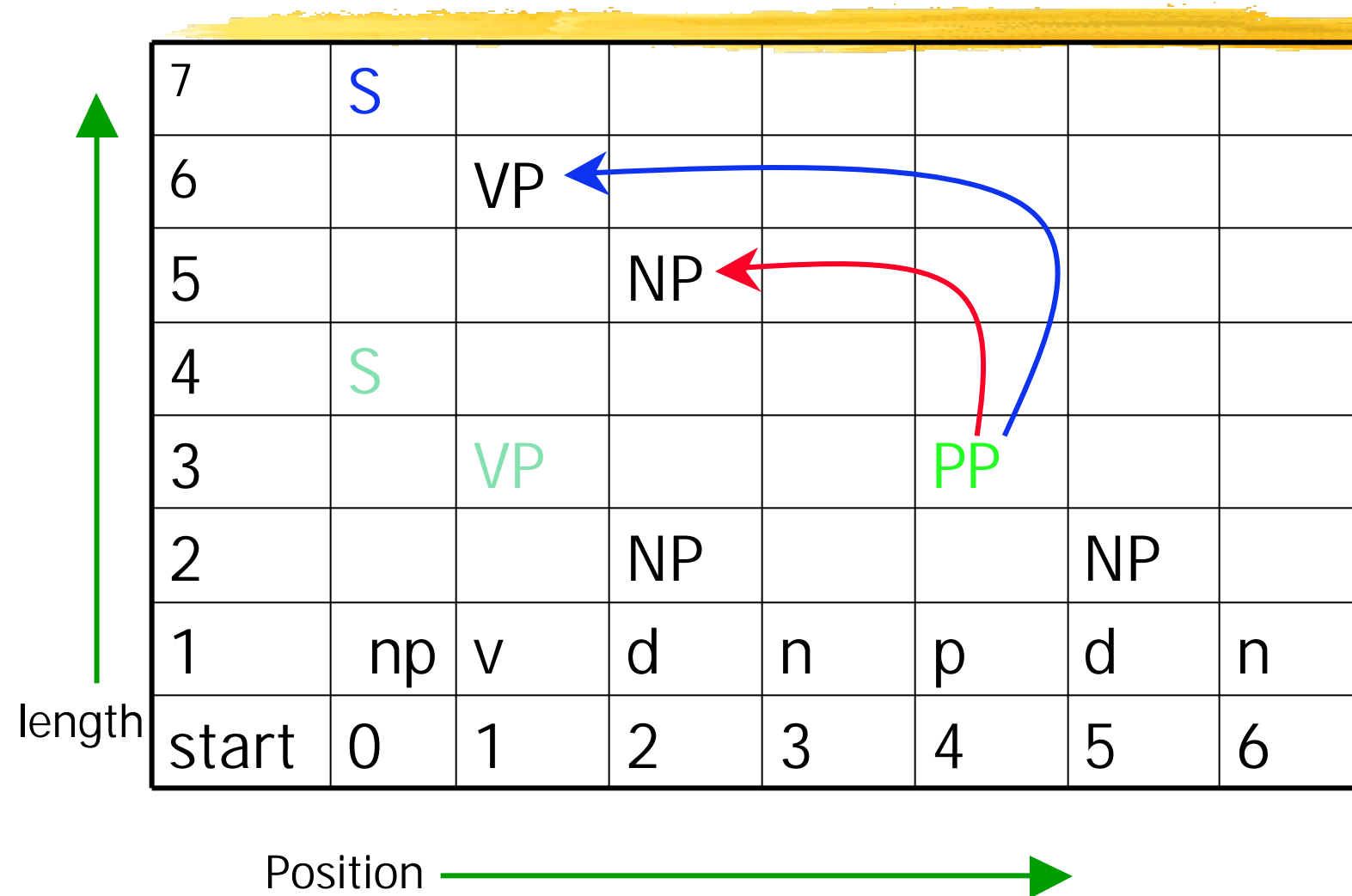
Chart in graphical form

7	S						
6		VP					
5			NP				
4	S						
3		VP			PP		
2			NP			NP	
1	np	v	d	n	p	d	n
start	0	1	2	3	4	5	6

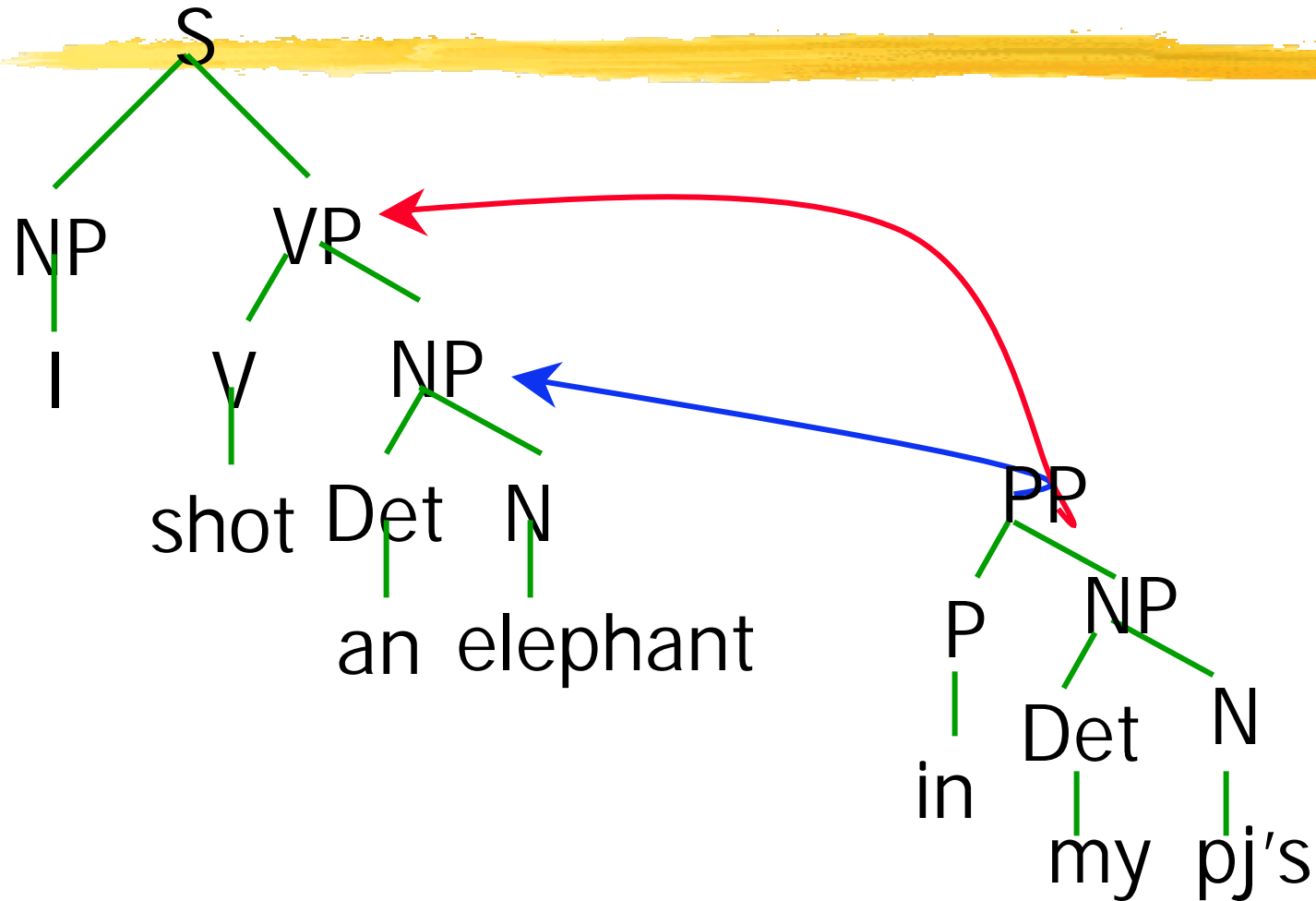
length ↑

Position →

At the end...



Corresponding to Marxist analysis



The chart



- A cell in the chart can contain more than one phrase (e.g., n & np)
- With each constituent is frequently stored information about which parsing rule was used to generate it and what smaller constituents make it up (to recover the parse)
- Used to prevent redundant work if 2 or more possible internal structures for a single phrase ("blue socks and shoes")

Edges

- Each edge consists of a grammar rule, plus info about how it matches up against the input, specifically:
 - A rule, e.g., $S(\text{entence}) \rightarrow NP\ VP$
 - The position up to which we've matched the rule to the input, indicated by a dot (\bullet), e.g., $S \rightarrow NP \bullet VP$
 - The starting position of the edge (the first input word matched) (e.g., VP 'shot...' starts at position 1)
 - The # of input words matched so far
- Edges organized by ending position (the last input word matching against their rule so far)
- Edges are *added* but never *deleted*

Edges, cont'd

Start

0 $S \rightarrow \bullet NP VP$
 $NP \rightarrow \bullet Det N$
 $NP \rightarrow \bullet N$
 $NP \rightarrow \bullet NP PP$

1 $NP \rightarrow N \bullet$
 $S \rightarrow NP \bullet VP$
 $NP \rightarrow NP \bullet PP$
 $VP \rightarrow \bullet V NP$
 $VP \rightarrow \bullet V NP PP$
 $PP \rightarrow \bullet P NP$

Etc...

State-set construction

Initialize: $S_0 \leftarrow$ initial state set = initial
state edge
 $[\text{Start} \rightarrow \bullet S, 0, n] \cup$
 ϵ -closure of this set under
predict, complete

Loop: For word $i=1, \dots, n$
 S_i computed from S_{i-1}
(using *scan, predict, complete*)
try *scan*; then *predict, complete*

Final: Is a final edge in S_n ?
 $[\text{Start} \rightarrow S\bullet, 0, n] \in S_n$?

The overall algorithm

- Suppose there are n words in the input
- Set up chart of height and width n
- Add input words onto stack, last word at bottom
- For each ending position i in the input, 0 through n , set up two sets, S_i and D_i ("Start", "Done")

$S_0 \leftarrow$ all rules expanding start node of grammar

$S_i \leftarrow \emptyset$ for $i \neq 0$

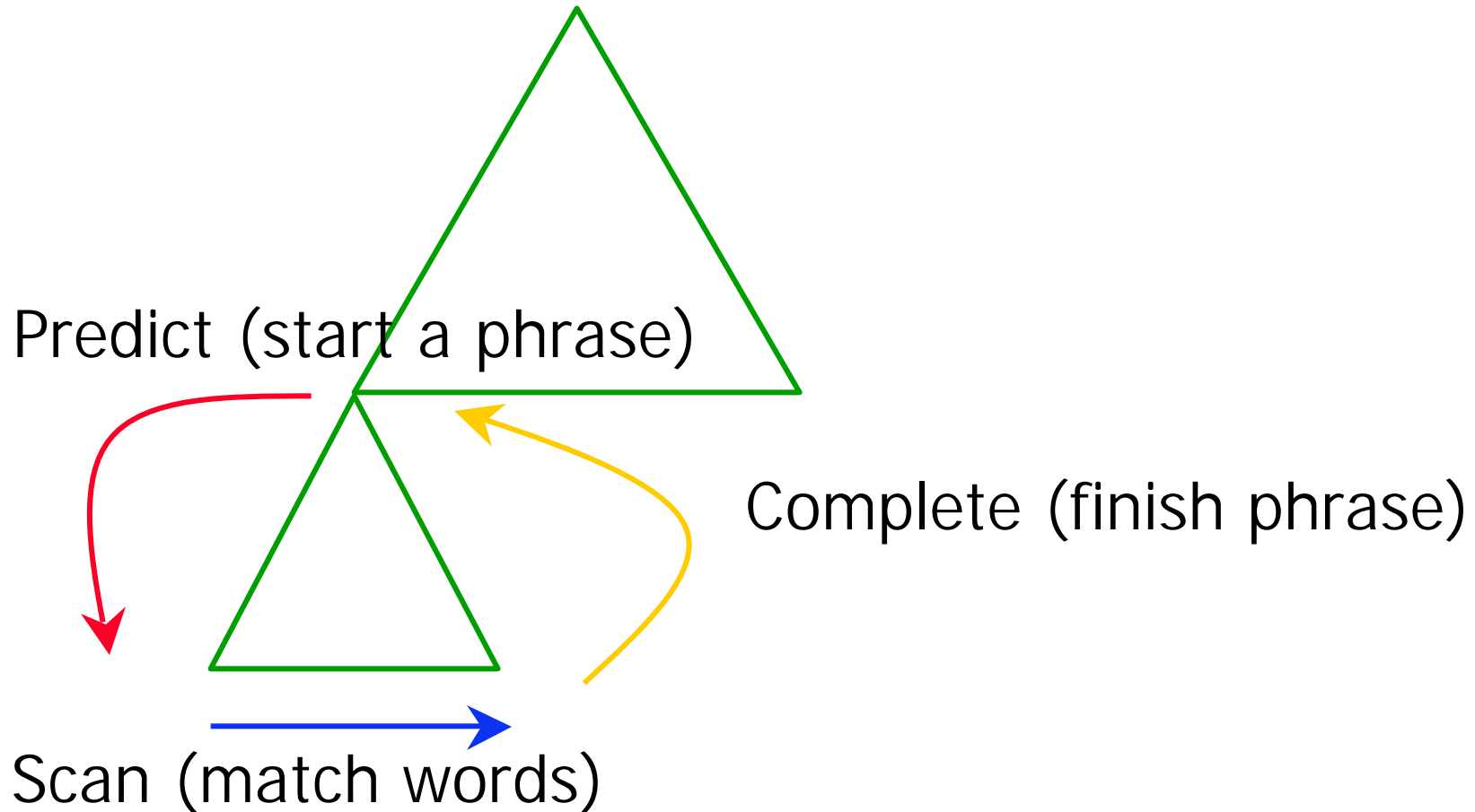
S_i will be treated as search queue (BFS or DFS).
Edges will be extracted one by one from S_i & put into D_i . When S_i becomes empty, remove 1st word from stack & go to next ending position $i + 1$

Or:

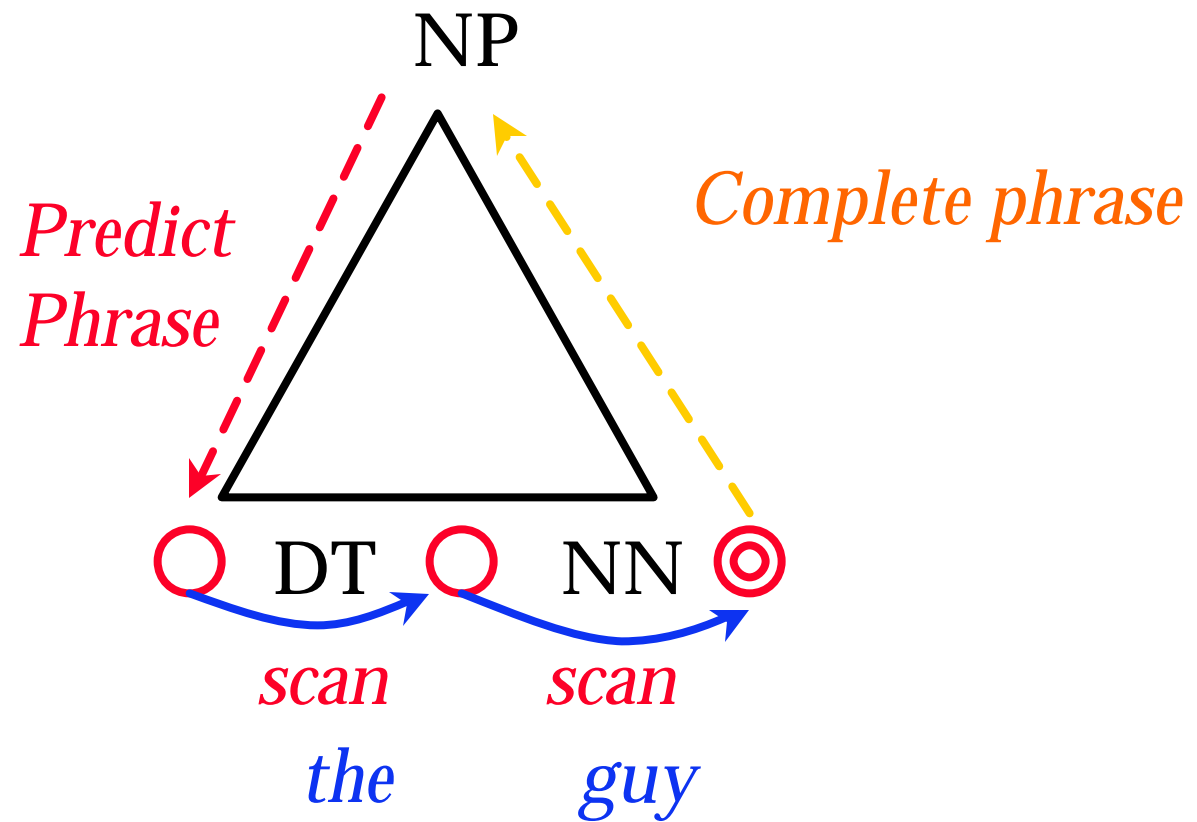
- Loop until S_i is empty
 - Remove first edge e from S_i
 - Add e to D_i
 - Apply 3 extension operations to e , using the 3 operators: scan, complete, predict (which may produce new edges)
 - New edges added to S_i or S_{i+1} , if they are not already in S_i , D_i , or D_{i+1}
- Pop first word off input stack
- When all ending positions processed, chart contains all complete phrases found

Adding edges by the 3 operations

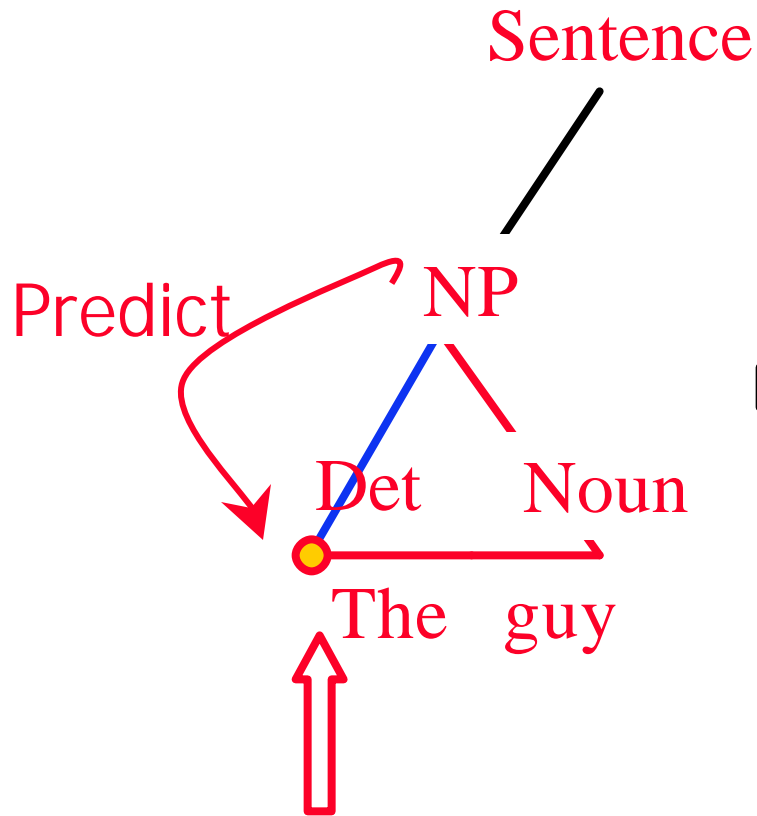
- like extending a search



Another way to view it



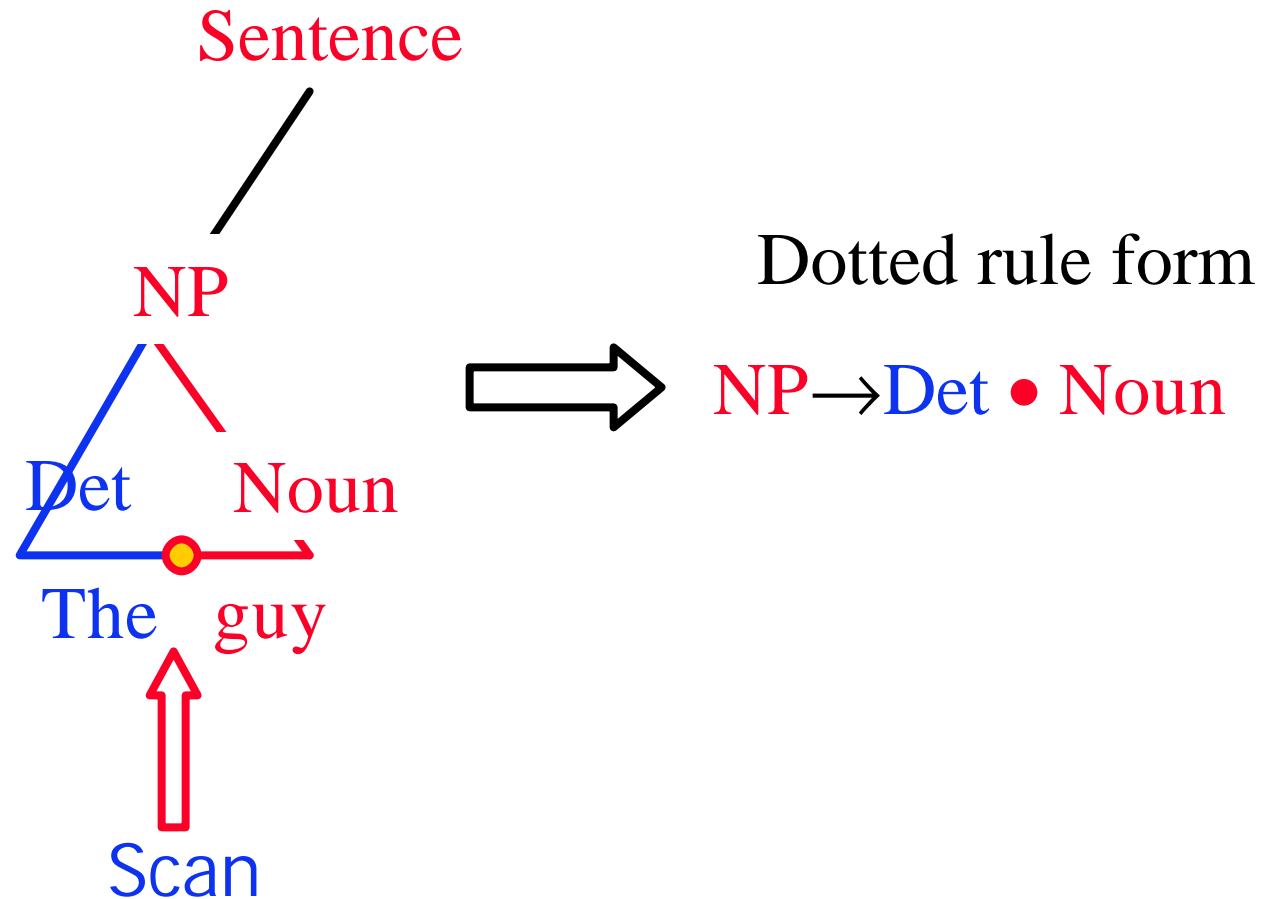
Connecting the dots



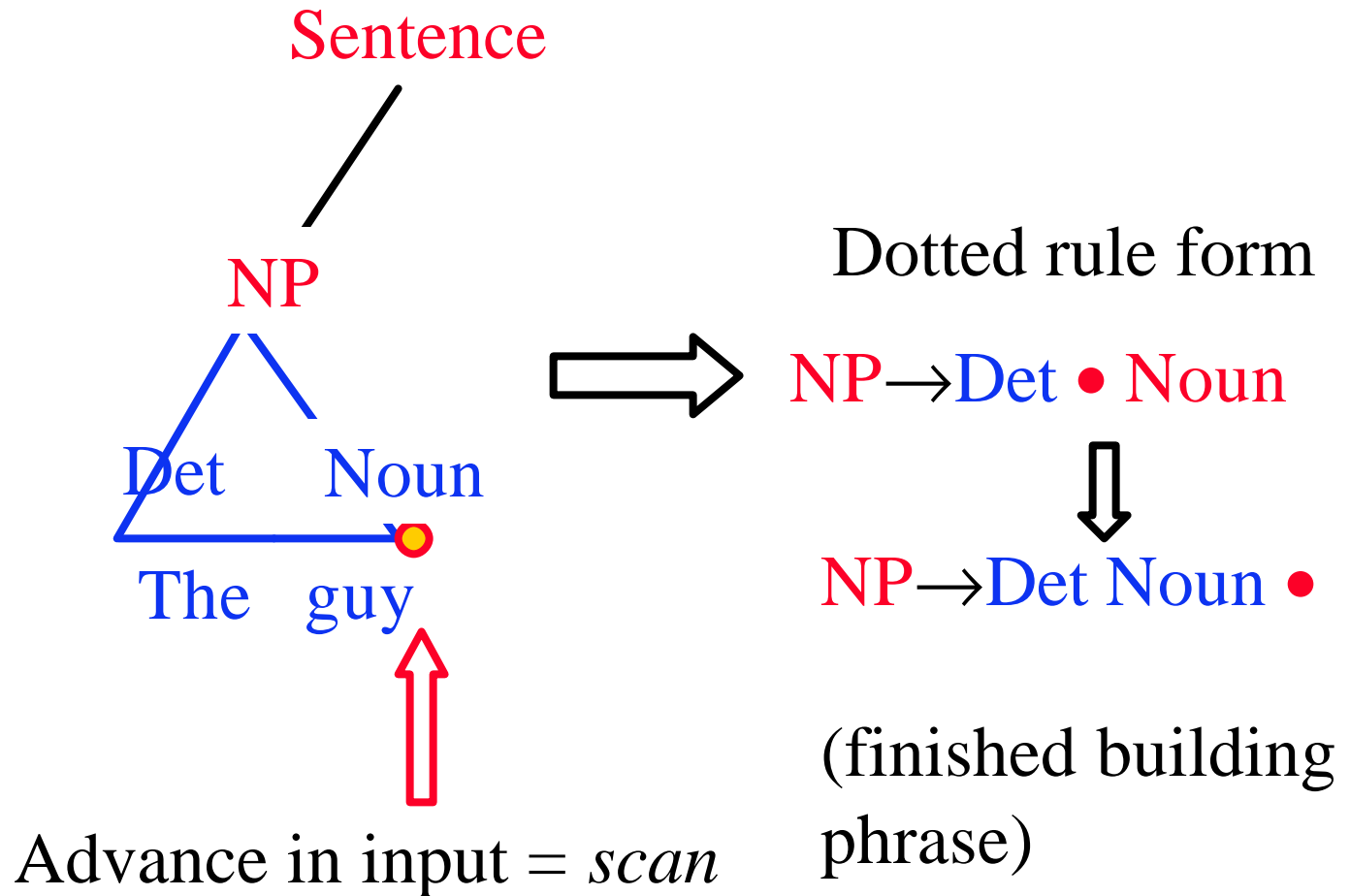
Dotted rule form
 $NP \rightarrow \bullet \text{Det Noun}$

Dot at beginning=
just started building a
phrase of this type

The dots – in the middle



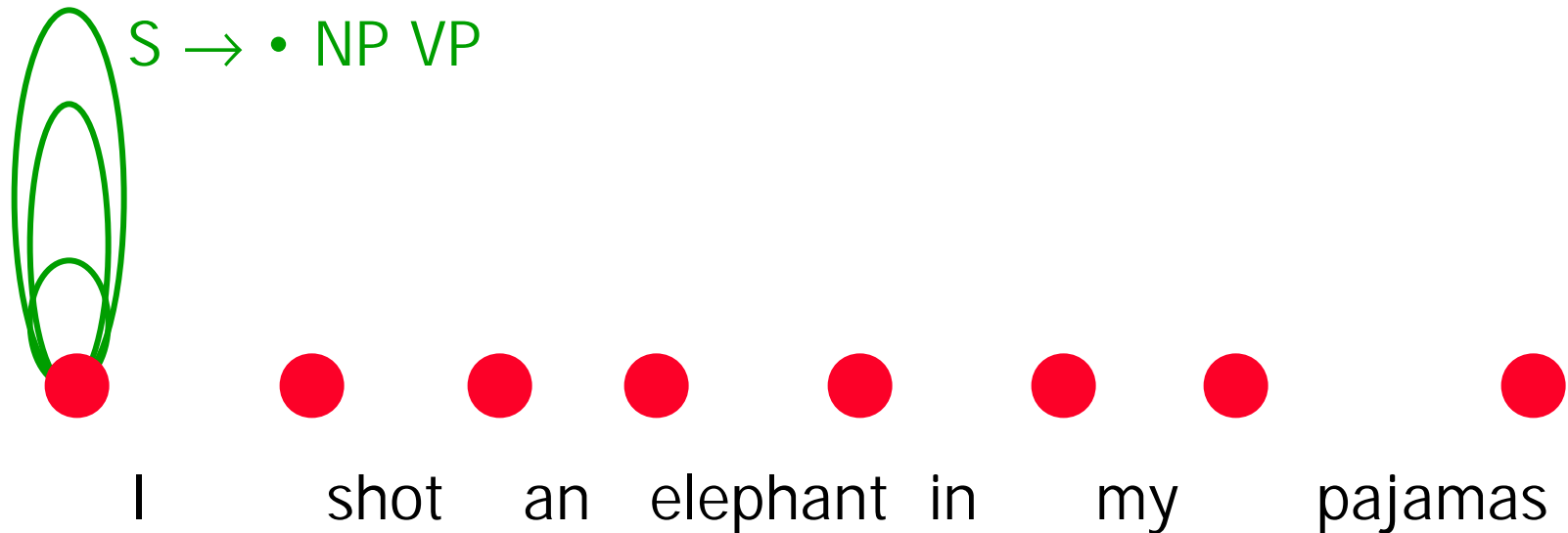
Dot at end



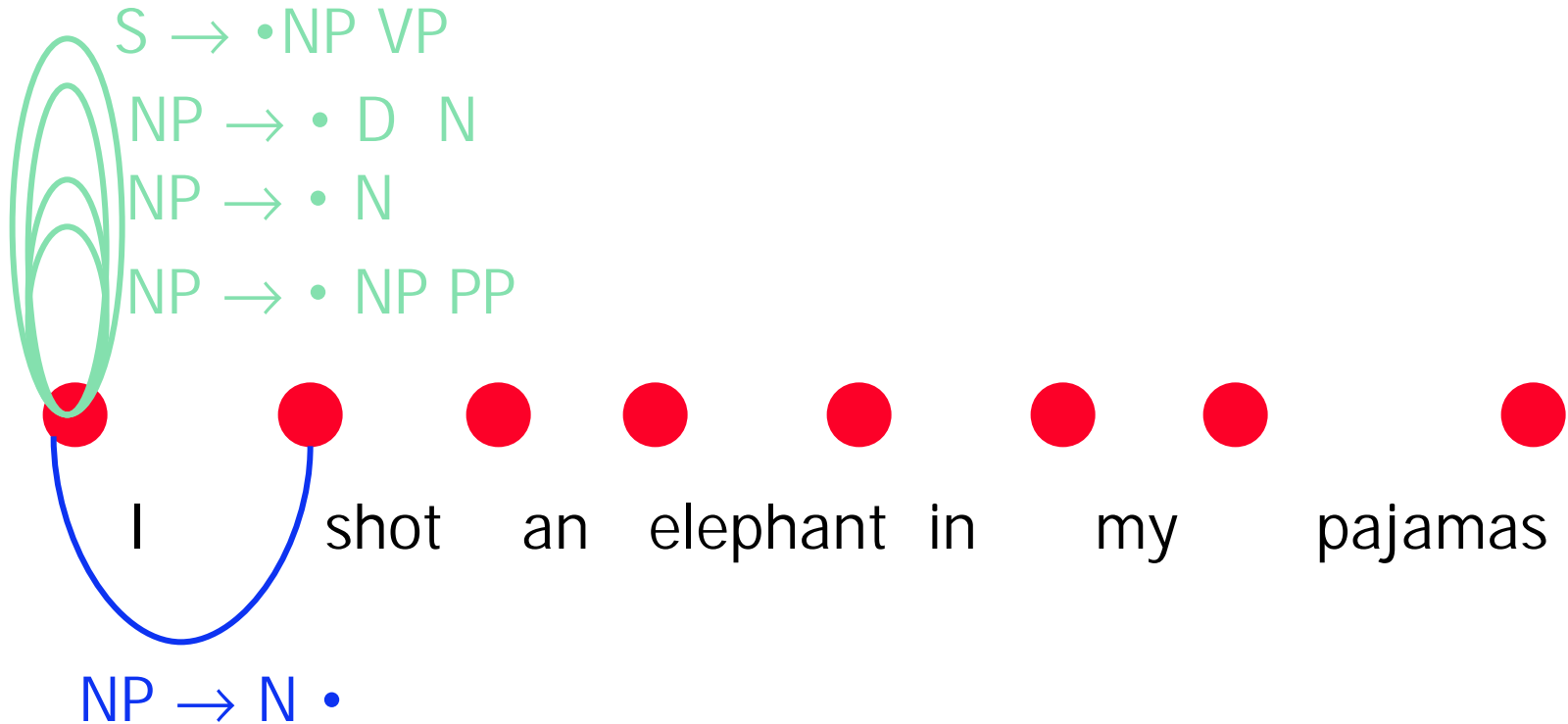
The three ops add edges in our full chart representation ...

- Loops (Predict) – start a phrase
- Skips (Scan) – build phrase
- Pastes – glue 2 edges to make a third, larger one (Complete) – finish a phrase

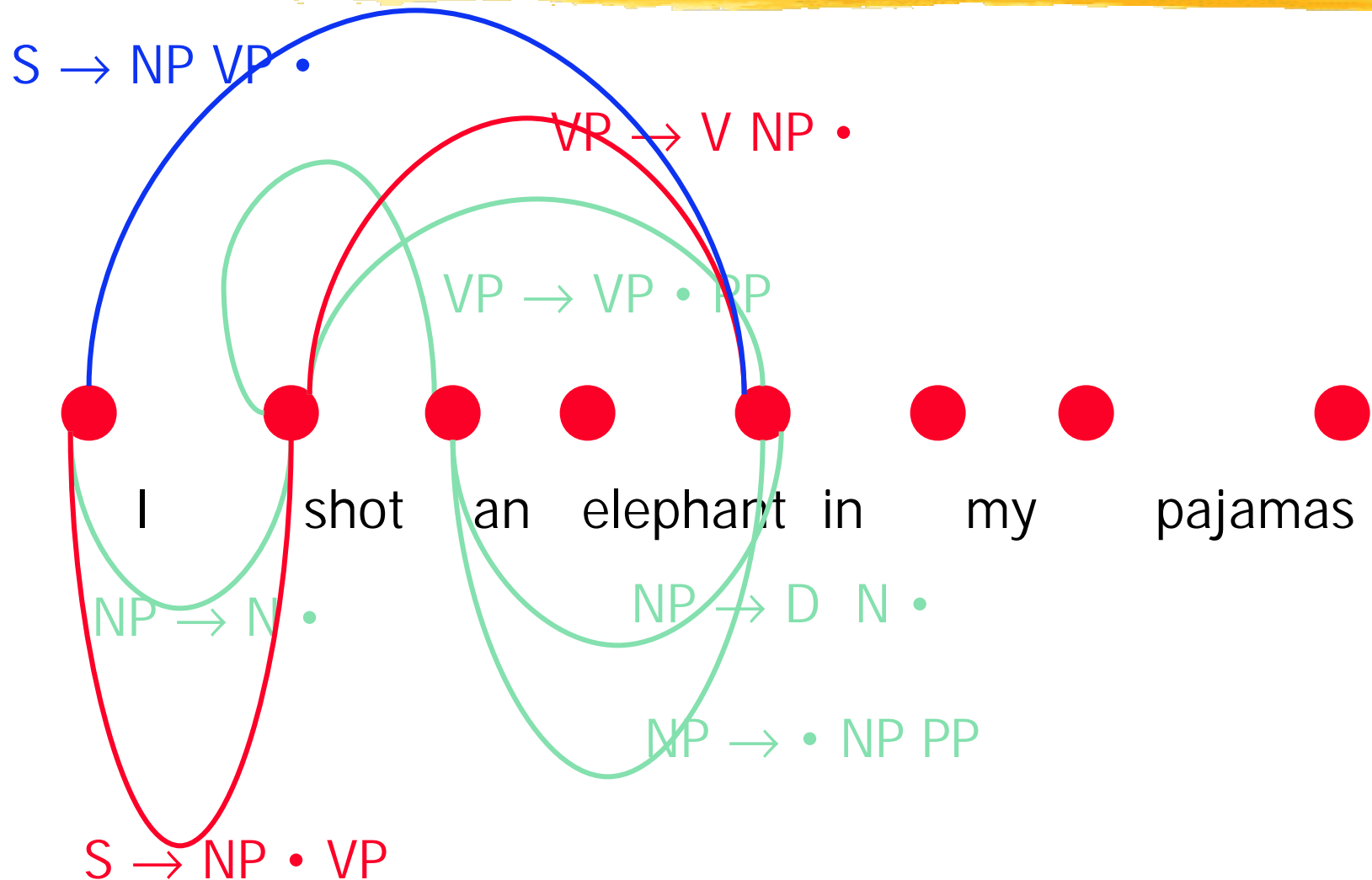
Picture: Predict adds the 'loops'



Picture: Scan adds the 'jumps'



Picture: Complete combines edges



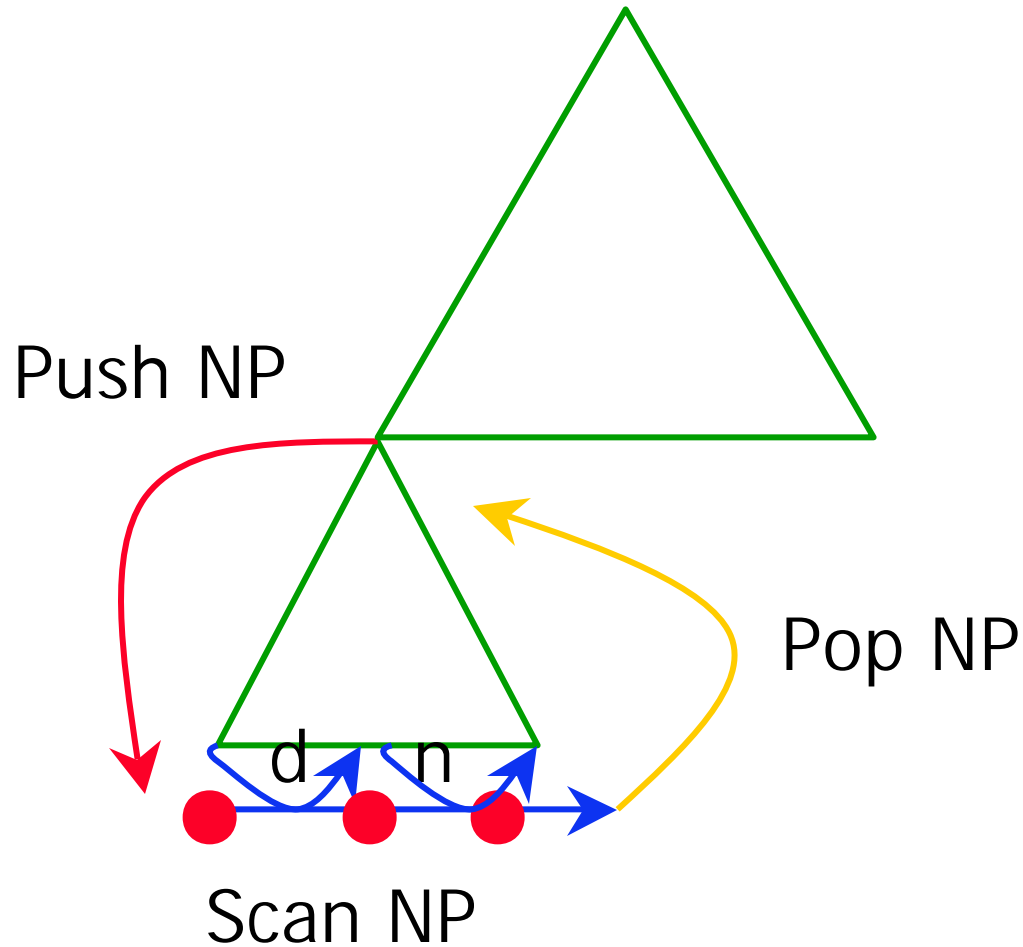
The ops

- 3 ops: *scan, predict, complete*; or
scan, push, pop
 1. *Scan*: move forward, consuming a token (word class) -
what if this is a *phrase name*, though?
 2. *Predict* (*push*): start building a phrase (tree) at this point
in the input; or jump to subnetwork;
 3. *Complete* (*pop*): finish building a phrase (tree) at this
point; pop stack and return from subnet (which also says
where the subphrase gets *attached*)

Scan = linear precedence;

Predict, complete: dominance

Another way to view it



Definitions – words & symbols



- Scan

Suppose current edge e is not finished & part of speech tag X follows the dot in the rule for e

Scan examines next word in input

If word has pos X , create new edge e' , identical to e except dot is moved one place to the right & length increment by 1

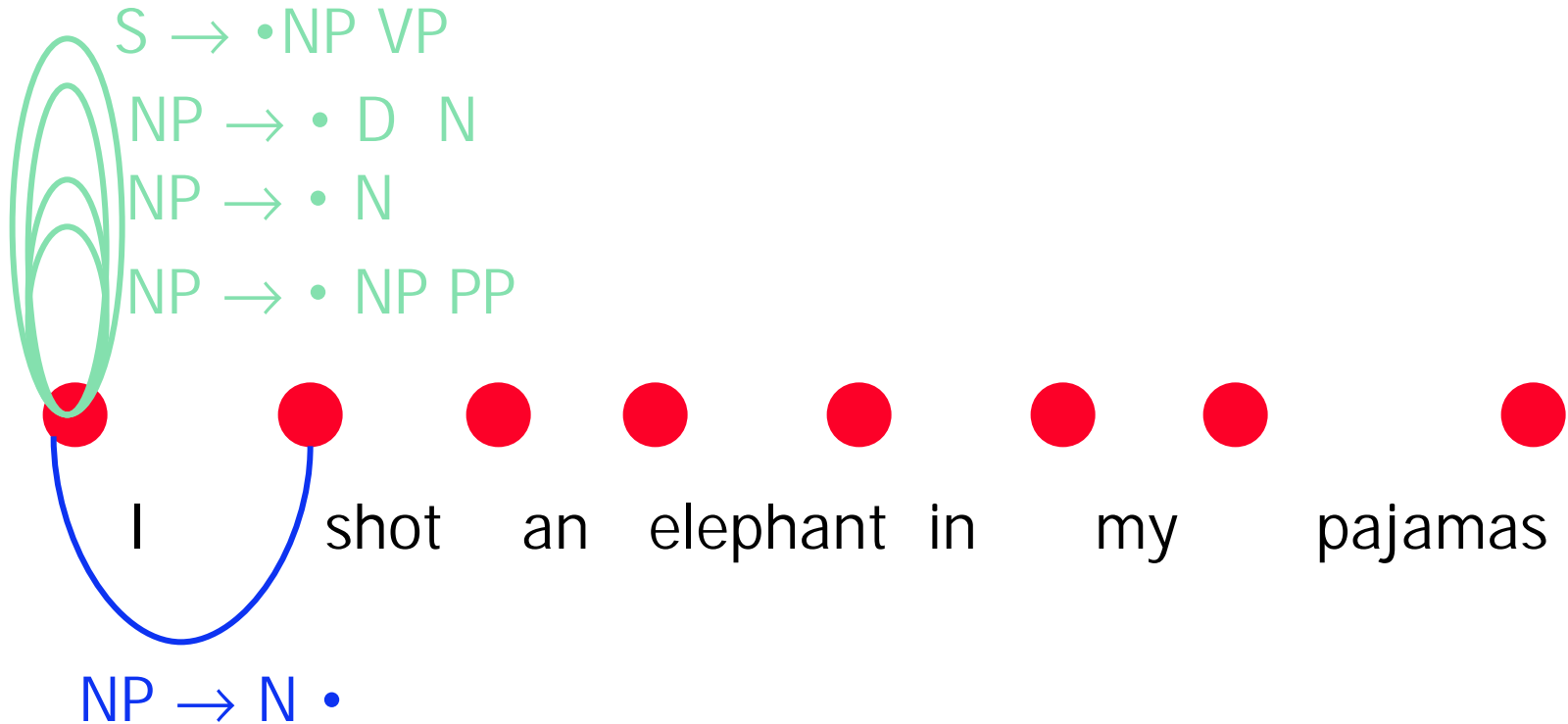
Add e' to S_{i+1}

Scan - formally



- Scan: (jump over a token)
 - Before: $[A \rightarrow \alpha \bullet t \beta, k, i]$ in State Set S_i & *word* $i = t$
 - Result: Add $[A \rightarrow \alpha t \bullet \beta, k, i+1]$ to State Set S_{i+1}

Picture: Scan adds the 'jumps'



Predict operation



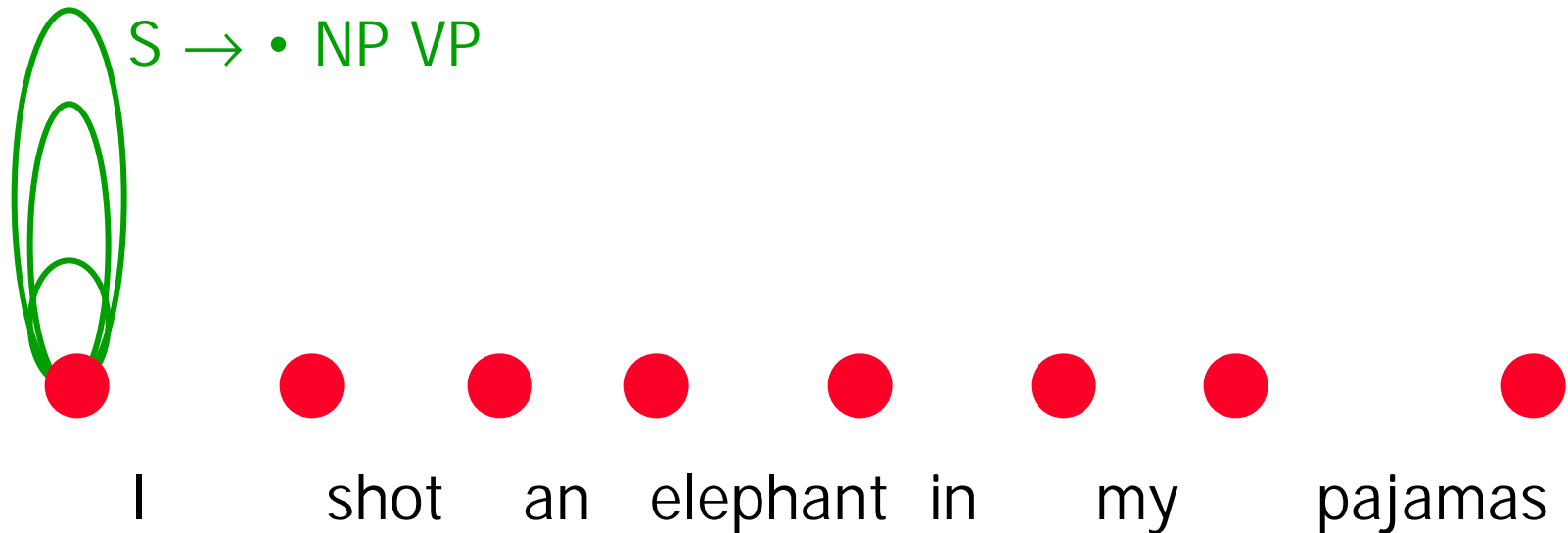
- Suppose current edge e is not finished
- Predict extracts next item X needed by e – the phrase after the dot in the edge
- Find all rules in grammar whose lefthand side is X
- For each of these, make a new edge with the dot on the left, and add edges to S_{i+1}

And again...



- Predict (Push):
 - Before: $[A \rightarrow \alpha \bullet B \beta, k, i]$, $B = \text{nonterminal}$, in S_i
then
 - After: Add all new edges of form $[B \rightarrow \bullet \gamma, i+1, i+1]$
to State Set S_{i+1}

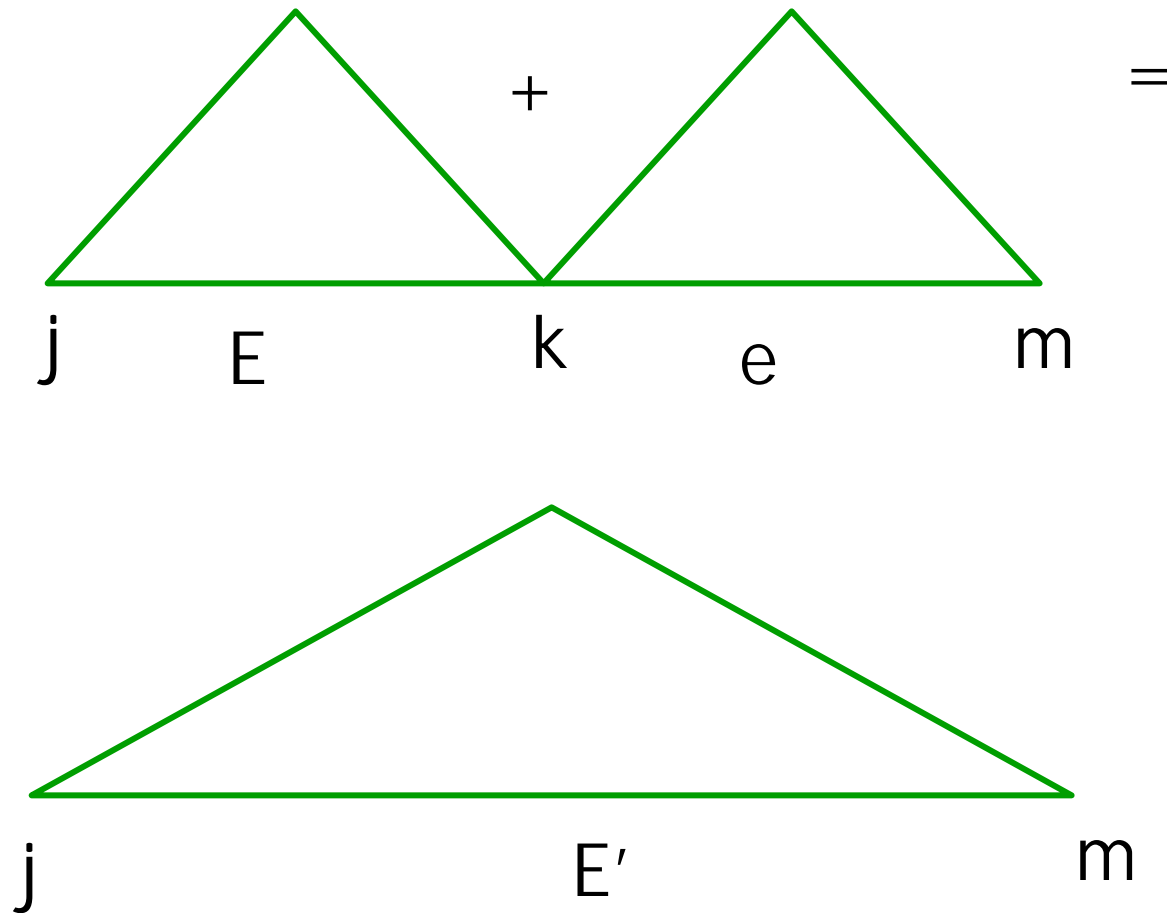
Picture: Predict adds the 'loops'



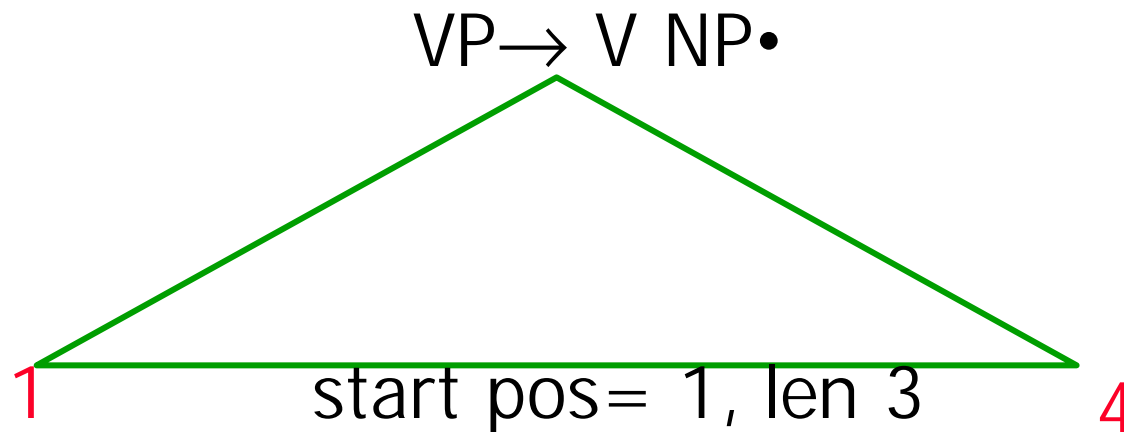
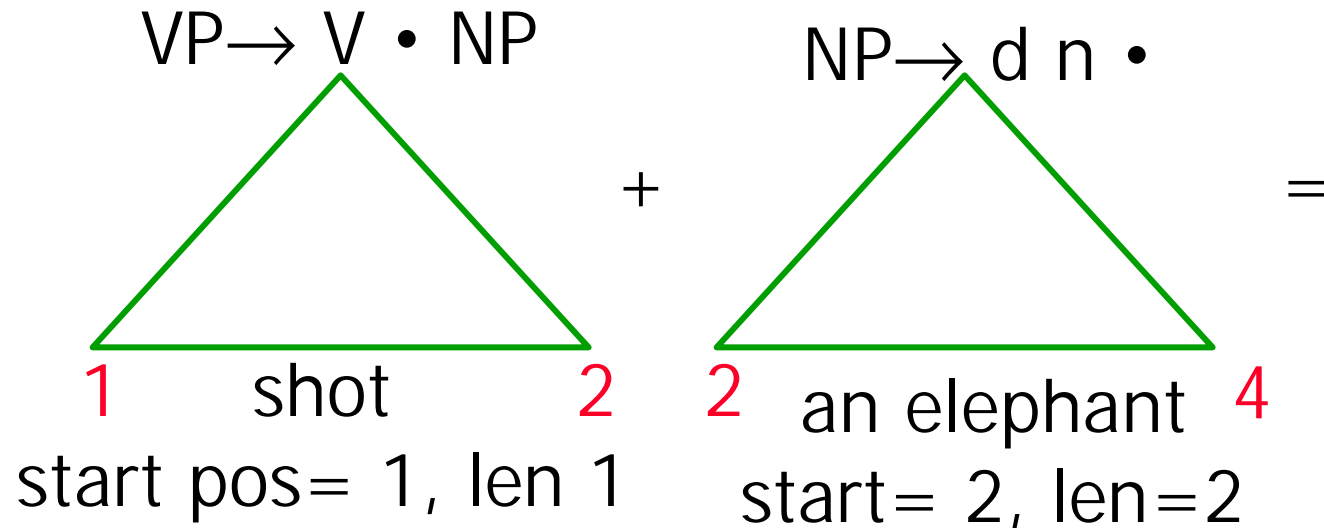
Complete (finish phrase)

- Suppose current edge e is finished (dot at right end). Suppose e looks like:
 $X \rightarrow y_1 y_2 \dots y_p \cdot$ from start pos k , length m
- Check if X is already in chart cell (k,m) . If so, add e to set of derivations for this phrase X .
- If X is not already in cell (k,m) then:
 - Examine each edge E in D_k . If E is incomplete, and the next item needed for E is X , create a new edge E' with dot hopped over X to the right
 - Length of E' is sum of lengths of $E + e$
 - Add E' to S_i

Picture of this – ‘pasting’ $X+Y$
together



"The fundamental rule"



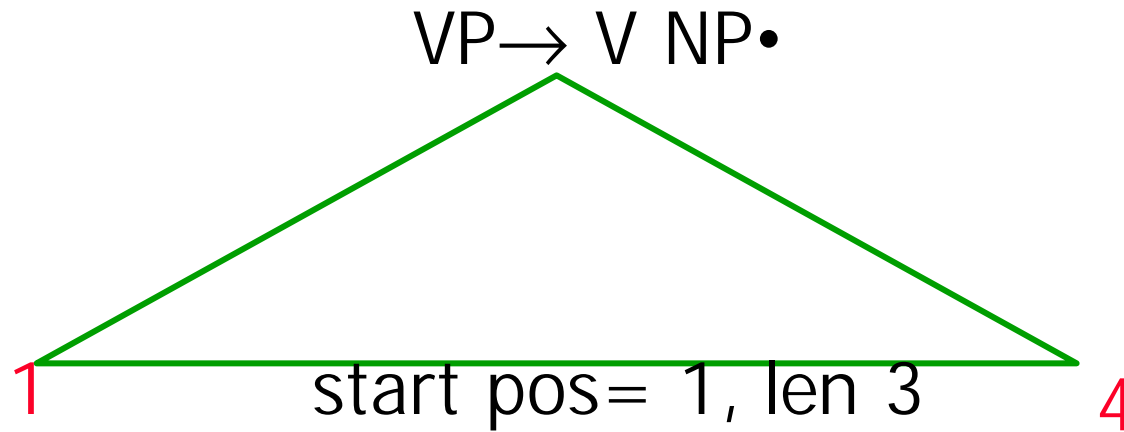
Adding to chart...

7							
6							
5							
4							
3		VP					
2			NP				
1	np	v	d	n	p	d	n
start	0	1	2	3	4	5	6

length ↑

Position →

This new edge E' will itself be processed... since dot is at end...



Go back to state set 1 & see what rule was looking for a VP

It's the rule $S \rightarrow NP \bullet VP \dots$ so we can paste these two subtrees together to get a complete S,
"I shot an elephant"

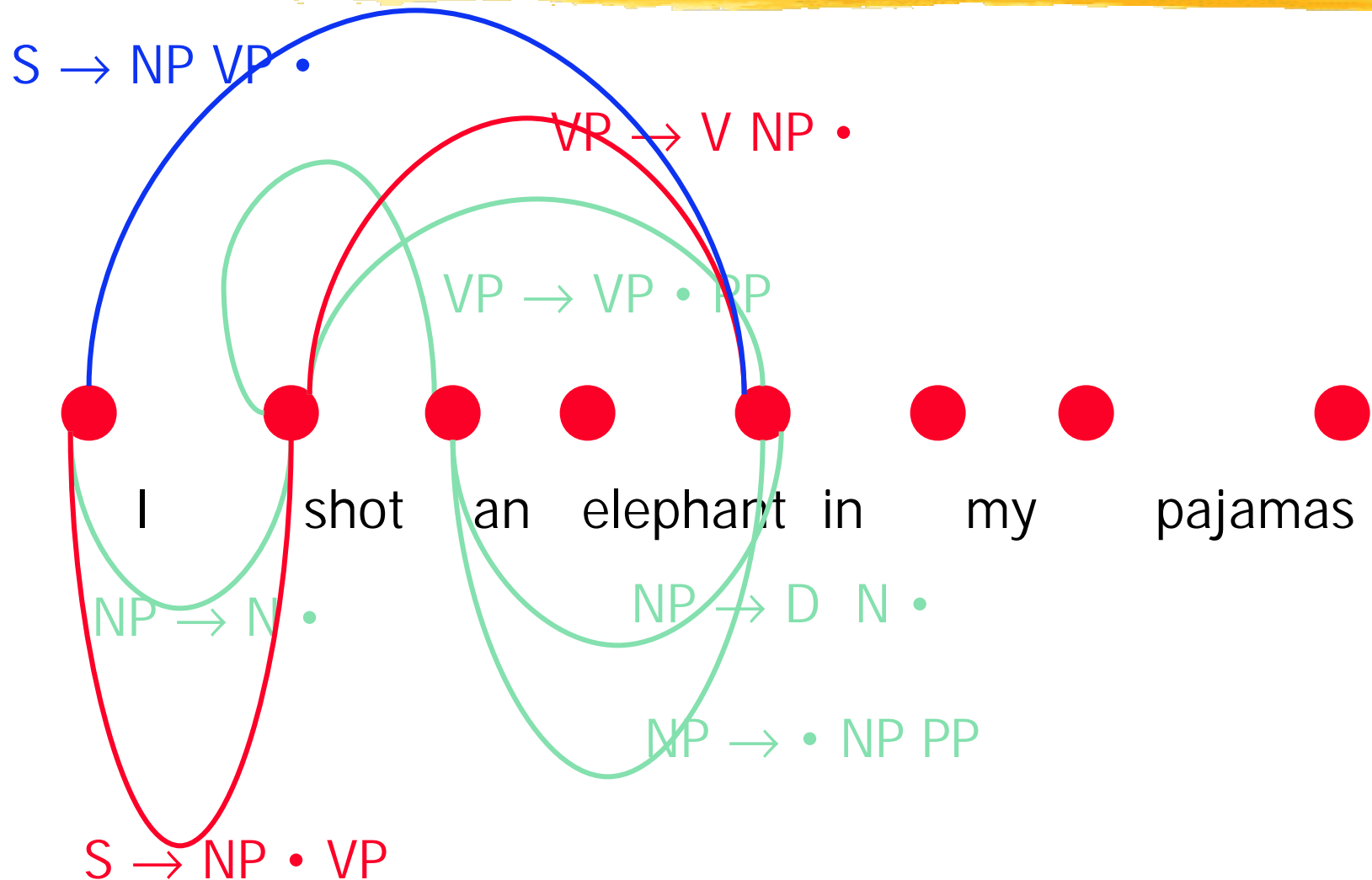
Adding the S

7							
6							
5							
4	S						
3		VP					
2			NP				
1	np	v	d	n	p	d	n
start	0	1	2	3	4	5	6

More precisely

- Complete(Pop): (finish w/ phrase)
- Before: If S_i contains e in form $[B \rightarrow \gamma \bullet, k, i]$ then go to state set S_k and for *all* rules of form $[A \rightarrow \alpha \bullet B \beta, k, j]$, add E' $[A \rightarrow \alpha B \bullet \beta, j, i]$ to state set S_i

Picture: Complete combines edges



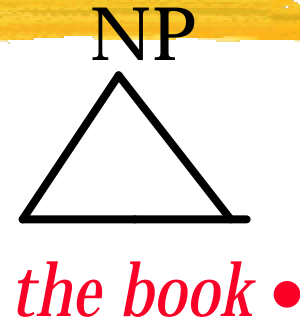
Scan examples

$\text{NP} \rightarrow \text{Det Noun} \bullet$

the book •



$[A \rightarrow \alpha tt' \bullet \beta'', k, i+1]$



Predict ('wish') example

$S \rightarrow \bullet NP VP$

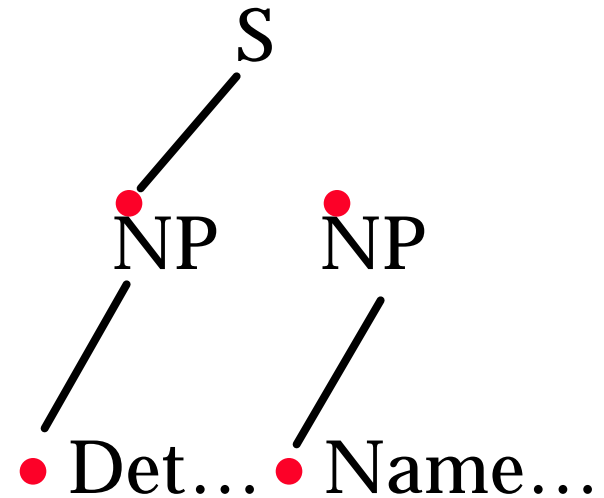
$NP \rightarrow \bullet Det Noun$

$NP \rightarrow \bullet Name$

$\bullet the\ book$
↑

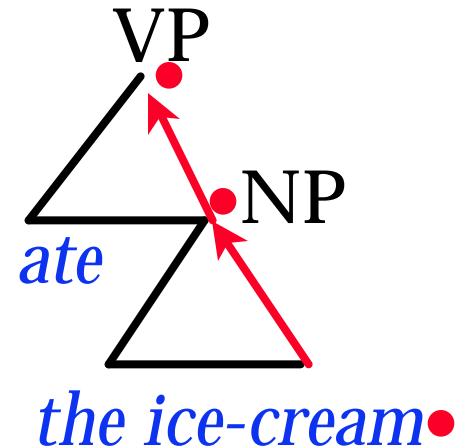
$A \rightarrow \alpha \bullet B \beta, k, i-1$

$B \rightarrow \bullet \gamma, i, i$



'Complete' example

VP → Verb NP • PP
VP → Verb NP •
NP → Det Noun •



...*ate the ice-cream* •

$[B \rightarrow \gamma \bullet, k, i]$

$[A \rightarrow \alpha B \bullet \beta, k, i]$

6.863J/9.611J Lecture 8 Sp03

...go back to previous
State Set & jump dot
(in *all* rules that called
for NP)

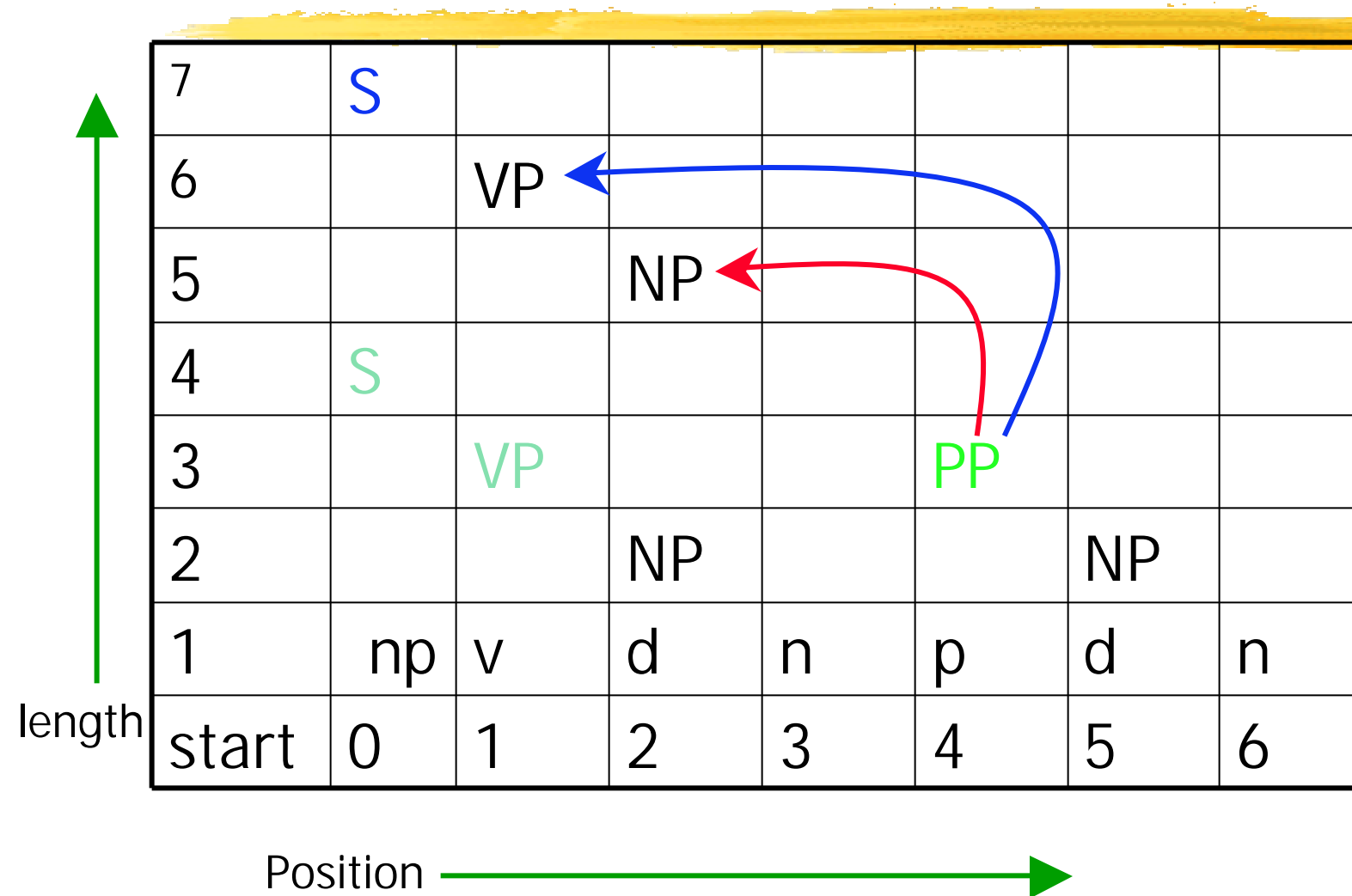
At the end..

7	S						
6		VP					
5							
4	S		NP				
3		VP			PP		
2			NP			NP	
1	np	v	d	n	p	d	n
start	0	1	2	3	4	5	6

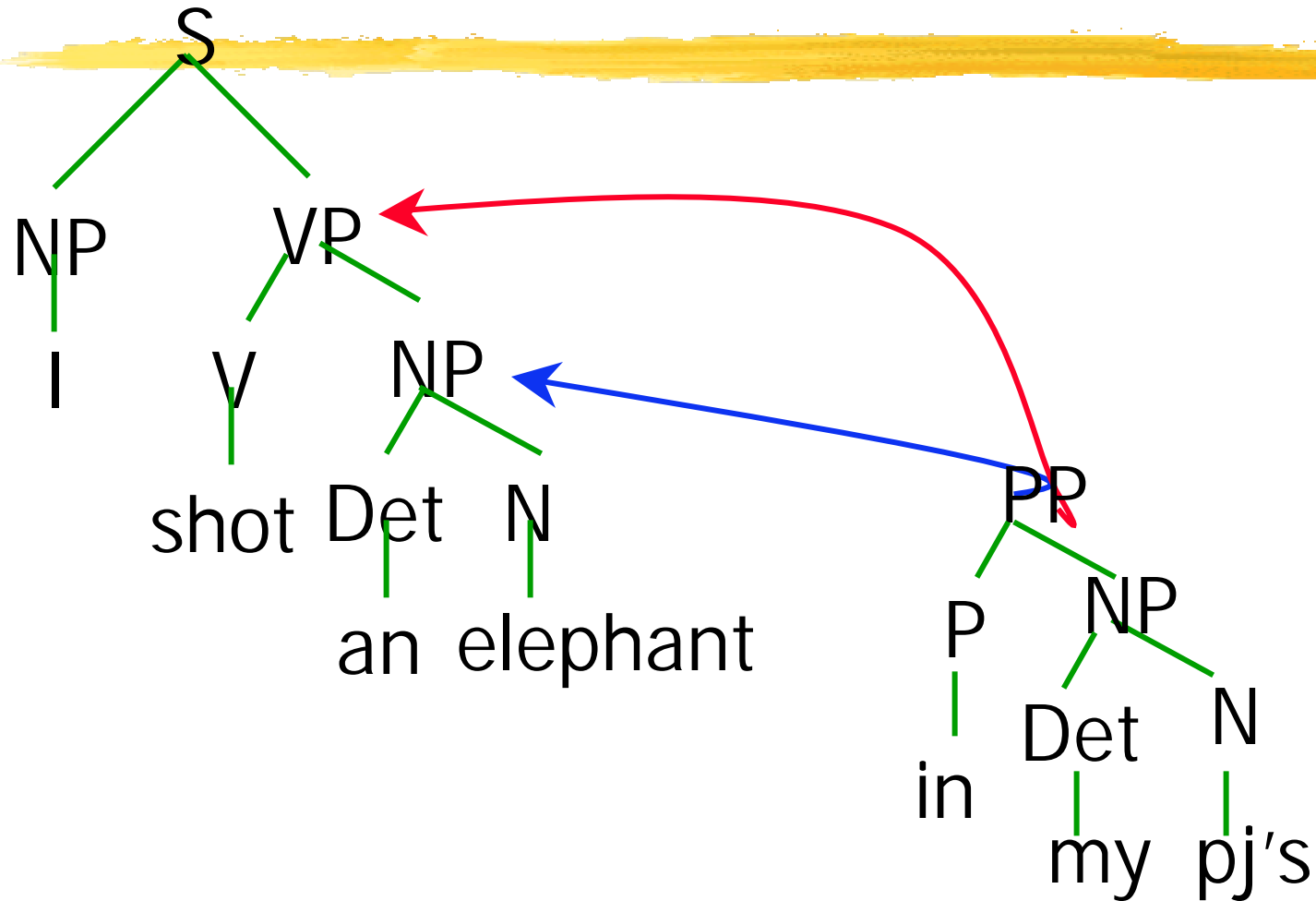
length ↑

Position →

At the end...



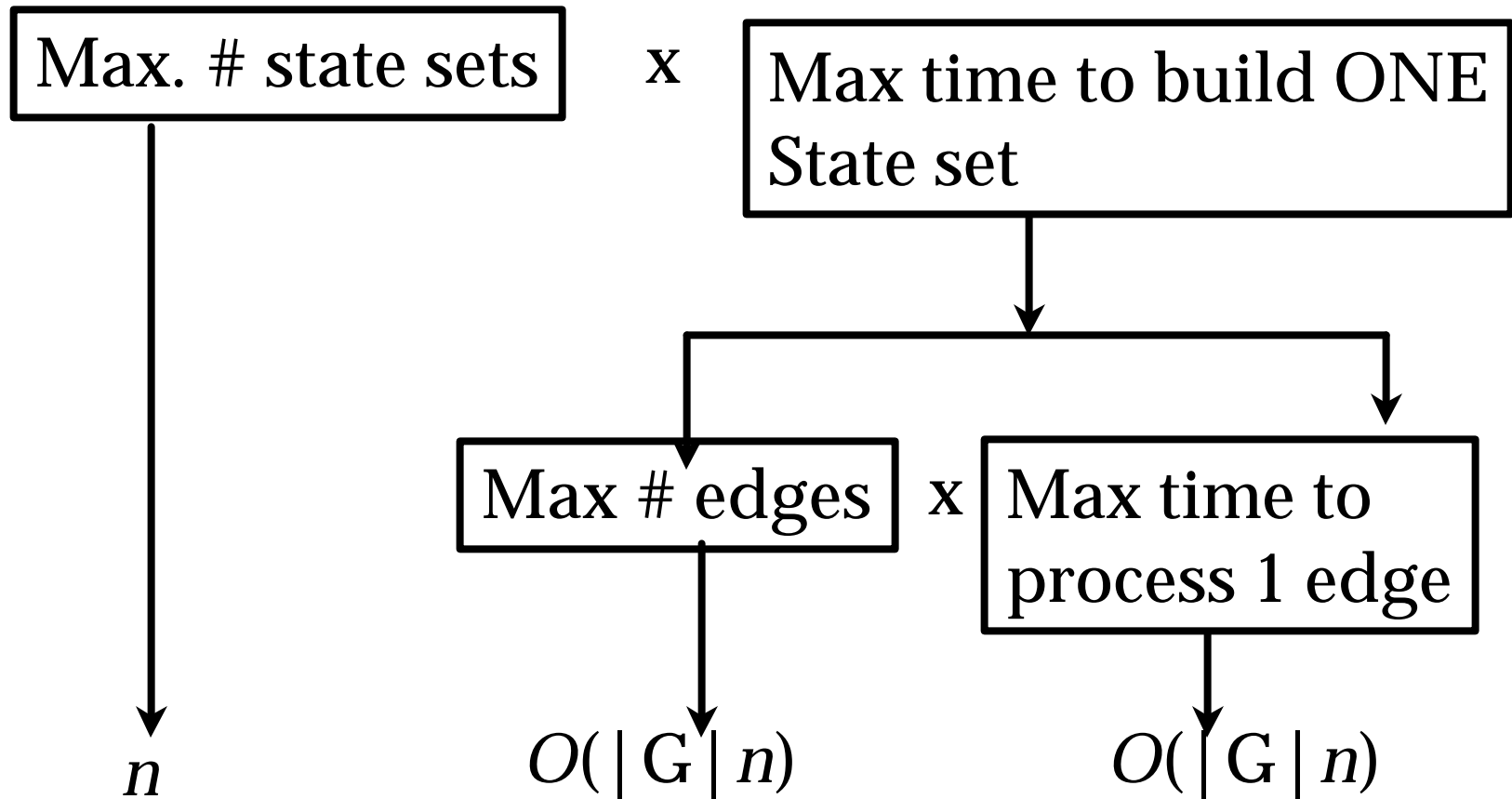
Corresponding to Marxist analysis



Please note:

- How ambiguity is handled
 - Multiple attachments, with dynamic programming principle: once we have built a PP spanning positions $[3, 7]$ *we use it twice*
 - This is the key to sub-exponential parsing: we don't have to *enumerate all the possibilities explicitly*
- Why we don't have to list identical items twice (another part of the same rule)
- For parsing, we use backpointers to keep track of *which* item causes a new item to be added - this gives us a chain for the state sequence = the path

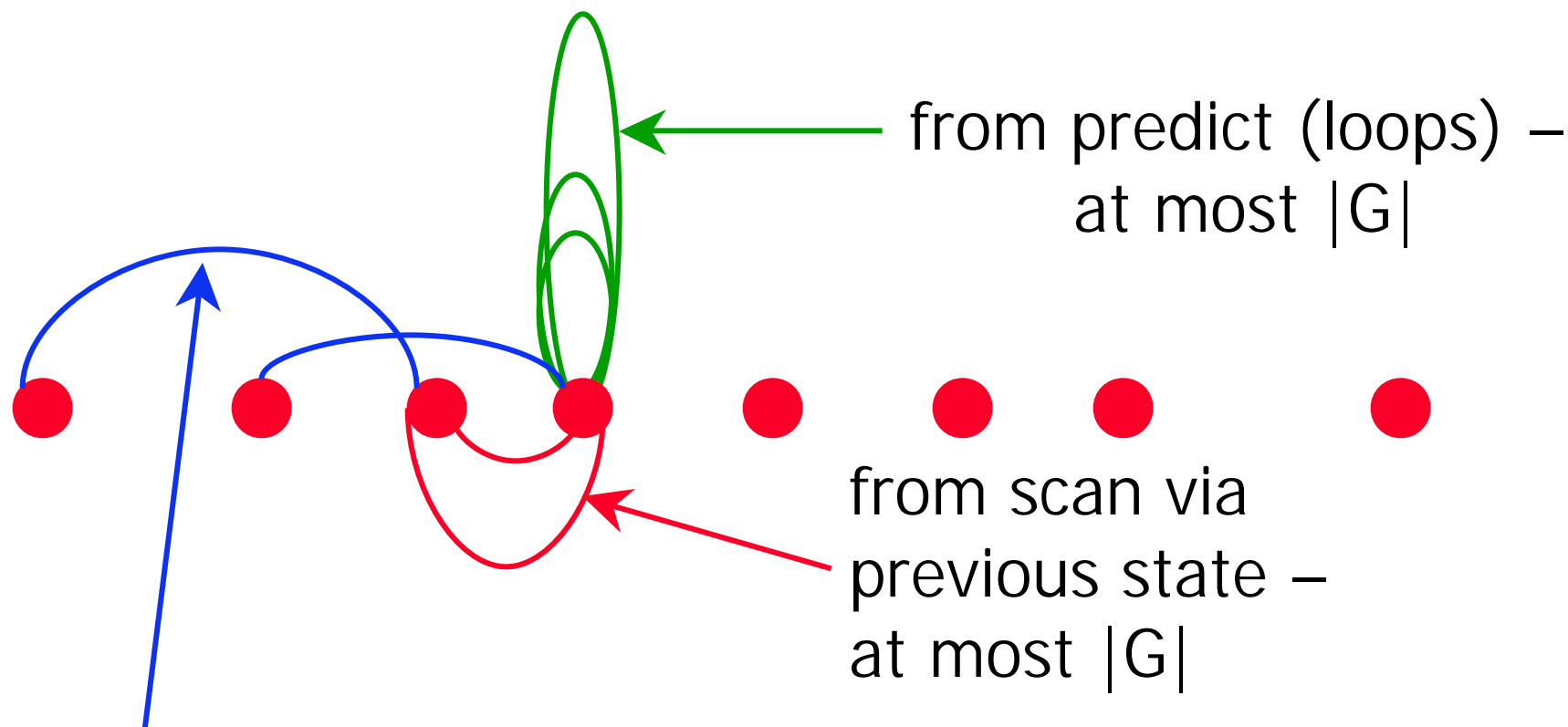
So time complexity picture looks like this:



Time complexity

- Decompose this in turn into:
 1. time to process a single edge in the set
 2. times maximum # distinct edges possible in one state set (assuming no duplicates!)
- Worst case: max. # of distinct edges:
 - Max # of distinct dotted rules x max # of distinct return values, i.e., $|G| \times n$
 - (Why is this?)
 - (Edges have form: dotted rule, start, len)
- Note use of grammar size here: amount of 'chalk' = Σ # symbols in G .

Max # distinct edges: loops, incoming from scans, incoming from paste:

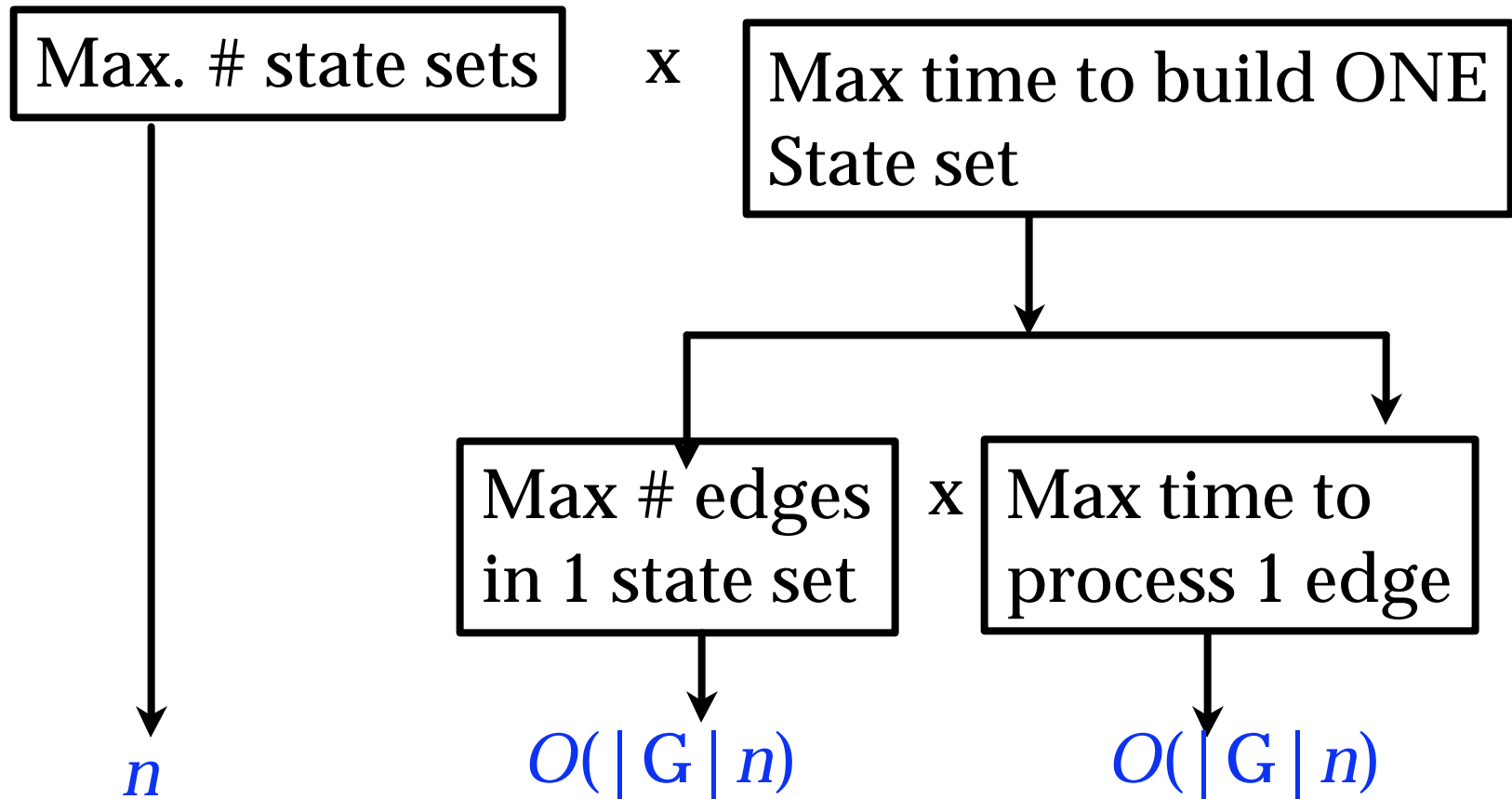


From complete – could come from *any* preceding state – at most $n \cdot |G|$

Time complexity, continued

- The time to process a single edge is found by separately considering time to process *scan*, *predict*, and *complete* operations
- Claim: *Scan*, *predict* constant time (in $|G|$ and n , n = length of sentence)
- Because we can build in advance all next-state transitions, given the Grammar
- Only action that takes more time is *complete*!
- For this, we have to go back to previous state set and look at *all* (in worst case) edges in *that* state set - and we just saw that in the worst case this could be $O(|G| \times n)$

So time complexity picture looks like this:



Grand total



- $O(|G|^2 n^3)$ - depends on *both* grammar size and sentence length (which matters more?)
- Lots of fancy techniques to precompute & speed this up
- We can extend this to optional elements, and free variation of the 'arguments' to a verb

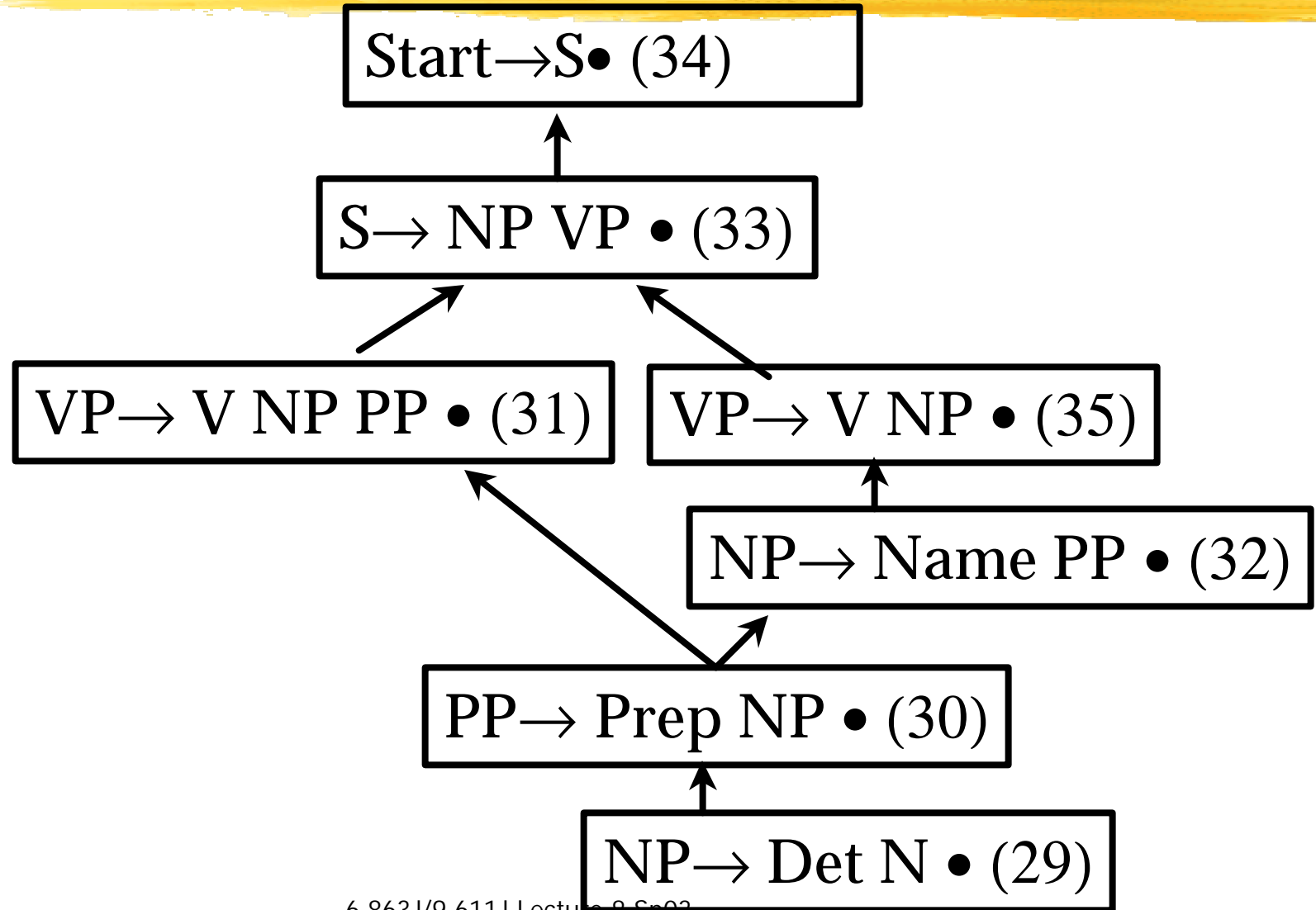
How do we recover parses?



State set pointer structure that represents both parses

- Just like *fruit flies like a banana*
- Keep multiple backpointers to keep track of multiple ways that we use a 'completed' item (a whole phrase)
- The actual backpointer structure looks something like the following (one can show that it takes just $\log(n)$ extra time to construct this)

Backpointer structure



Recovering parses (structure, state sequence)

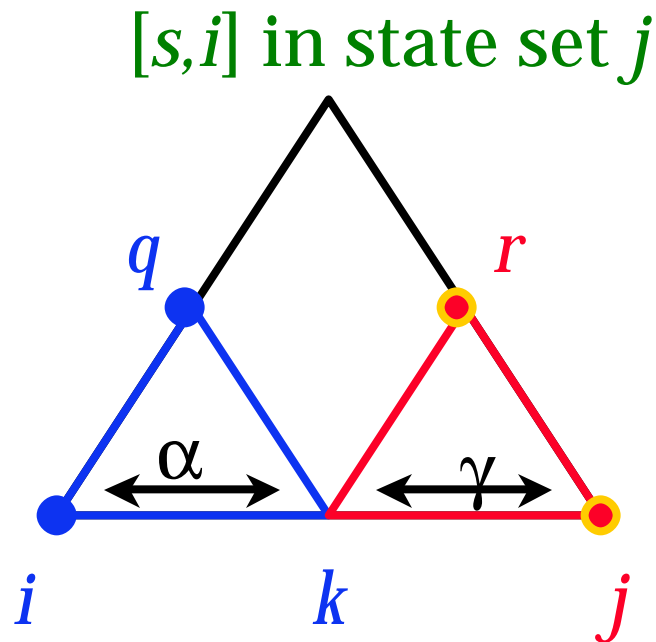
- Two basic methods: *online* & *offline*
- Simple offline method can build parse ptrs for all possible parses in time $O(n^3)$ – key is to build a ‘pruned’ collection of items (triples) in the state sets
- Why do we want to prune the state sets?
- Many items ‘die out’ because they fit the first part of an input sentence, but not the rest: e.g., *I think that blocks the light*
- Here we predict an NP for *that* and an NP for *that blocks* – one or more might die out.

Recovering parses



- Since semantic interpretation routines will run on syntactic structure and these are often more costly (why?) we want to reduce false paths ASAP

Simple queue algorithm to do this, based on the notion of 'useful' items- those that actually cause others to get added



- Any item in final state set is useful:
- If item $s=[A \rightarrow \alpha \bullet B, i]$ is in state set k & useful
- then item $q=[A \rightarrow \alpha B \bullet, k]$ & item $r=[B \rightarrow \gamma \bullet, j]$ are useful

Let $[s,i]$ denote an item with a dotted rule s & return pointer i .

Algorithm for recovering parses

[Initialize] Mark all items in state set S_n in the form $Start \rightarrow \alpha S \bullet, 0$

[Loop] *for* $j=n$ *downto* 0 *do*
 for $i=0$ *to* j *do*
 for every marked $[s,i]$ in state set j *do*
 for $i \leq k \leq j$, *if*
 $[q,i] \in S_k$ &
 $[r,k] \in S_j$ &
 $s = q \otimes r$ *then*
 mark $[q,i]$ and $[r,k]$

This is called a 'parse forest'



- Exponential # of paths, but we avoid building them all explicitly – we can recover any *one* parse efficiently

Worst case time for Earley algorithm

- Is the cubic bound ever reached in artificial or natural languages?
- Here is the artificial 'worst case' - # of parses arbitrarily large with sentence length; infinite ambiguity
- Here is the grammar:
 $S \rightarrow SS, SS \rightarrow a$
 $\{a, aa, aaa, aaaa, \dots\}$
- # of binary trees with n leaves =
 $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, \dots =$

$$\frac{1}{(n+1)} \binom{2n}{n}$$

Does this ever happen in natural languages?

- It does if you write cookbooks... this from an actual example (from 30M word corpus)

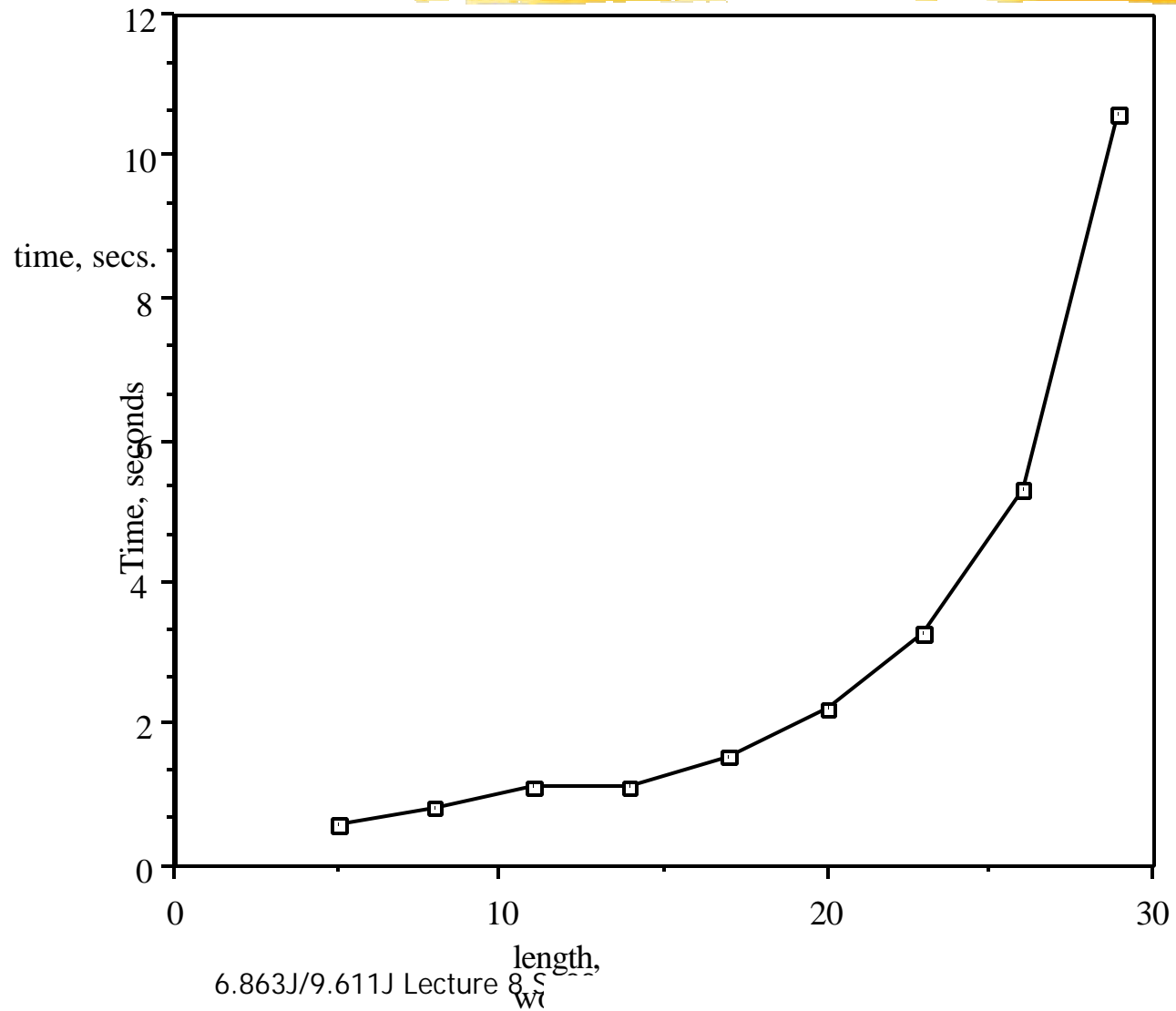
Combine grapefruit with bananas, strawberries and bananas, bananas and melon balls, raspberries or strawberries and melon balls, seedless white grapes and melon balls, or pineapple cubes with orange slices.

parses with 10 conjuncts is 103, 049
(grows as $6^{\# \text{conjuncts}}$)

This does indeed get costly -Verb NP PP example

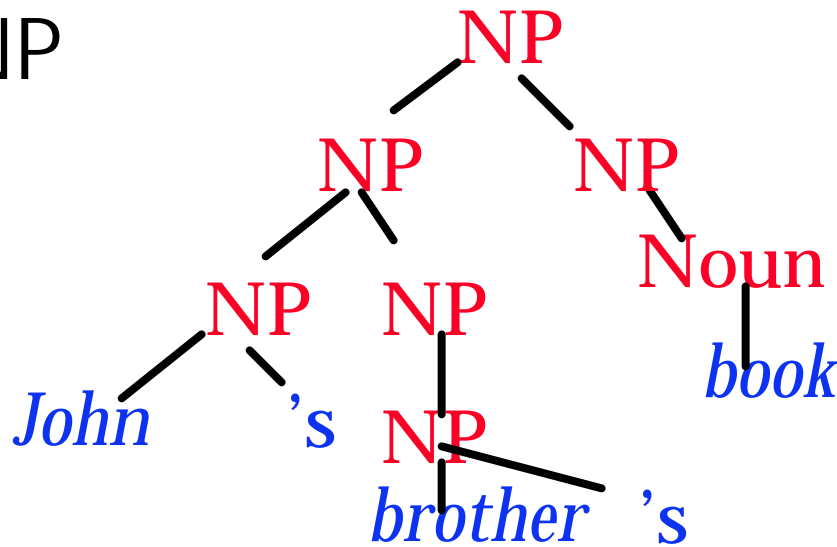
NP-PP ambiguity sentences NP P NP...

with Earley's algorithm



How this algorithm is clever

- Handling left-recursion
- Tail-recursion trick
- Example: *John's brother's book*
- $NP \rightarrow NP\ NP \mid NP \rightarrow \text{Noun} \mid \text{Noun 's}$
- Note how this *loops* on endless call to NP



...but *predict* cuts off after 1 round!

Note tail recursion

- State set S_0 :
- Add triples: $[NP \rightarrow \bullet \text{Noun}, 0, 0]$
 $[NP \rightarrow \bullet NP \ NP, 0, 0]$ *predict:*
 ~~$?[NP \rightarrow \bullet NP \ NP, 0, 0]$~~ ... No need!

Duplicate!

Note tail recursion: the call returns to itself – so no need to ‘keep’ return addresses in stack!

The edge loops to itself:



$[NP \rightarrow \bullet NP NP, 0, 0]$



Anything else?

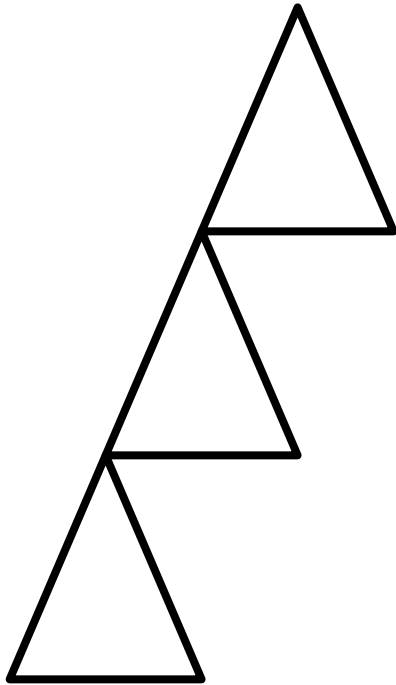


- If anything, Earley parsing is *too good* – it gets all the parses, even ones that people do not
- We shall see how to deal with this, using probabilities on rules; and
- Other parsing methods
- But first, what do people do?
- Consider the examples

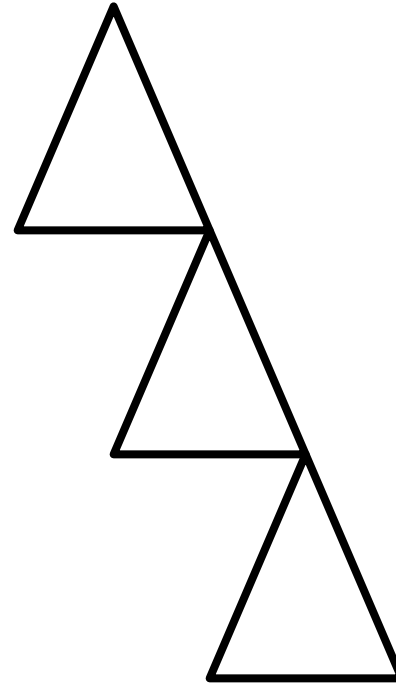
Wouldn't matter so much – but it
does seem to match what people do

- Both left- and right- branching 'structures' seem to be readily parseable by people without any sort of memory load (all other things being equal)
- *John's brother's mother's aunt....*
- *I believed that Mary saw that Fred knew that Bill shoveled snow*

Pictures of this..

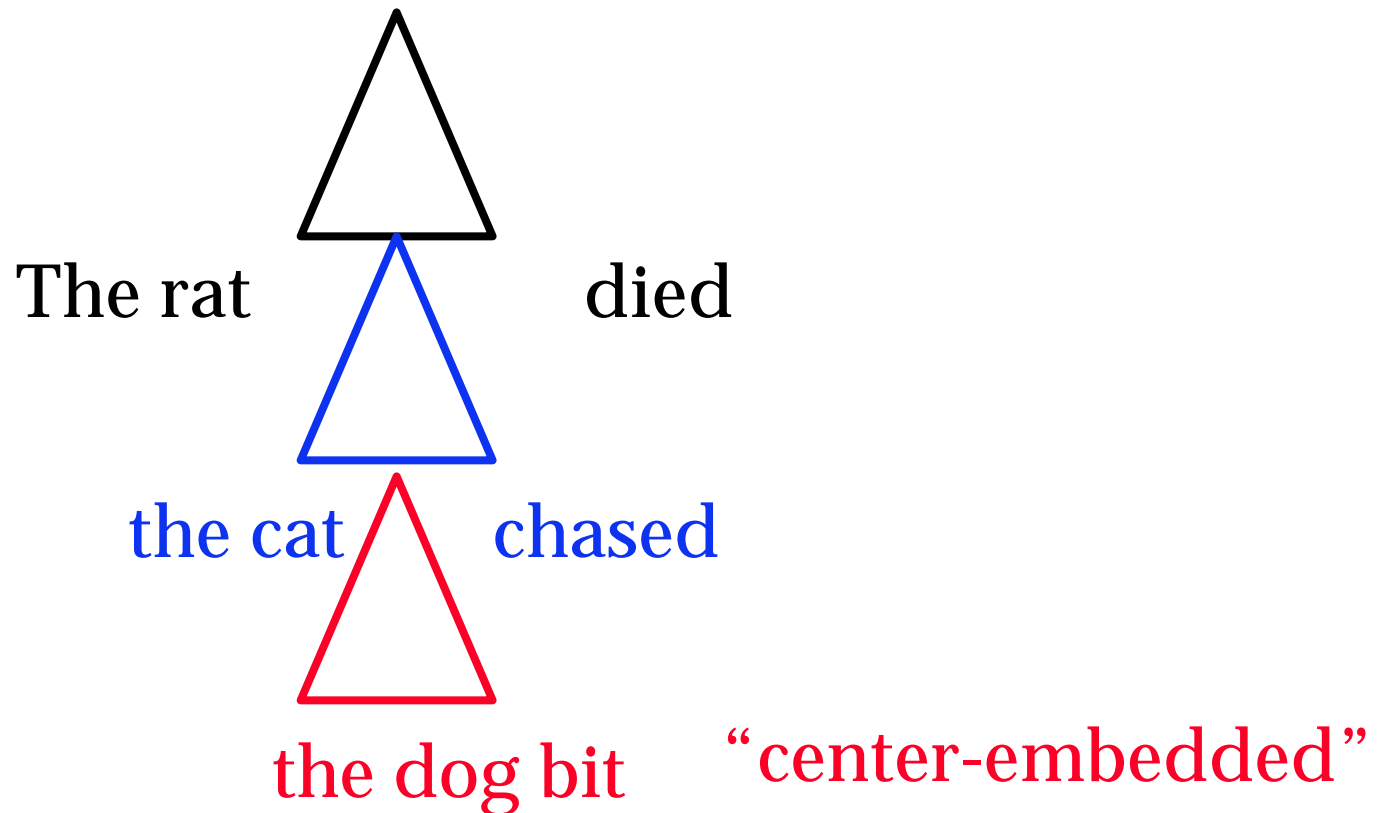


John's brother's book



*saw that Fred knew that
Bill shoveled snow*

So what's hard for people to process?



Why is this hard?



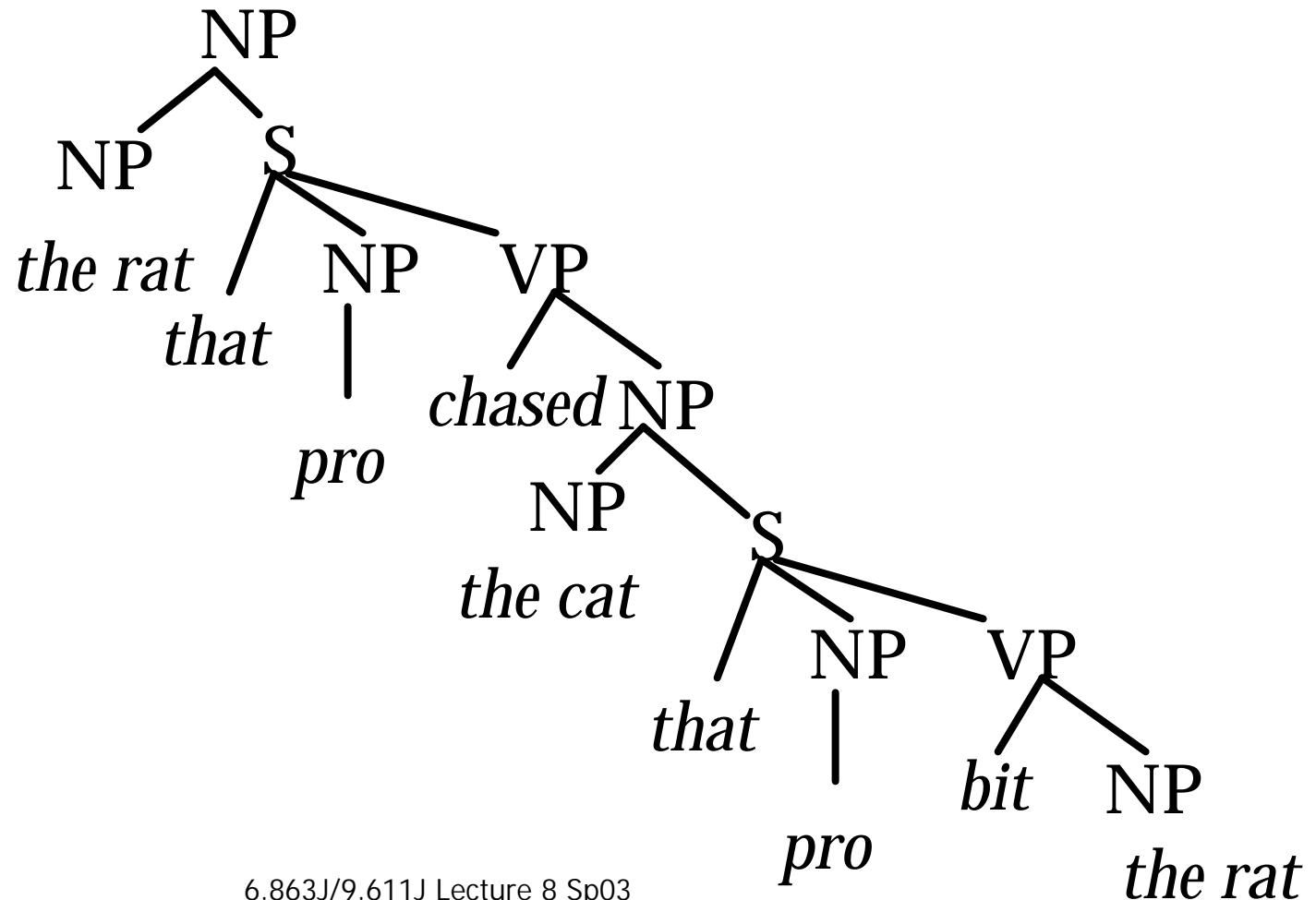
- Model: people have to “hold onto” open predicates (definition: open if verb+arguments have not yet been put together)
- In the preceding example, we have to hold onto a stack of Subjects (the rat, the cat, the dog...) before the corresponding verbs are seen
- This even shows up in unexpected places – speech intonational pattern actually seems to transduce center-embedded structures into left- or right- branching ones

Chomsky & Miller, 1959-63 analysis

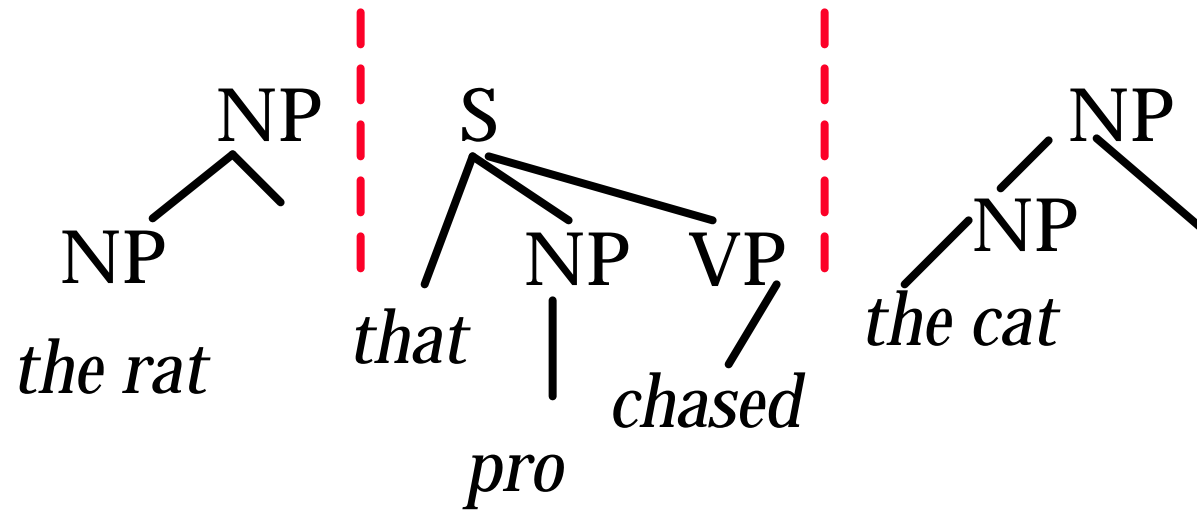
the dog that chased the cat that bit the rat

Parsing vs. intonational contours

Syntactic structure is center-embedded:



But the intonational structure follows this:



Suggests 2-stage parser (proposed by C&M):

Stage 1: parse into 'flat' structure

Stage 2: make 2nd pass & rearrange hierarchically

Also hints at how to do semantic interpretation – akin to syntax-driven translation

- Recall from compilers: if we *complete* the right-hand side of a rule, we can now *fire off* any associated semantic action (because we now have the item and all its 'arguments')
- This amounts to getting left-most complete subtree at each point to interpret
- Example:

$VP \rightarrow V \ NP$ • , e.g., "ate the ice-cream"

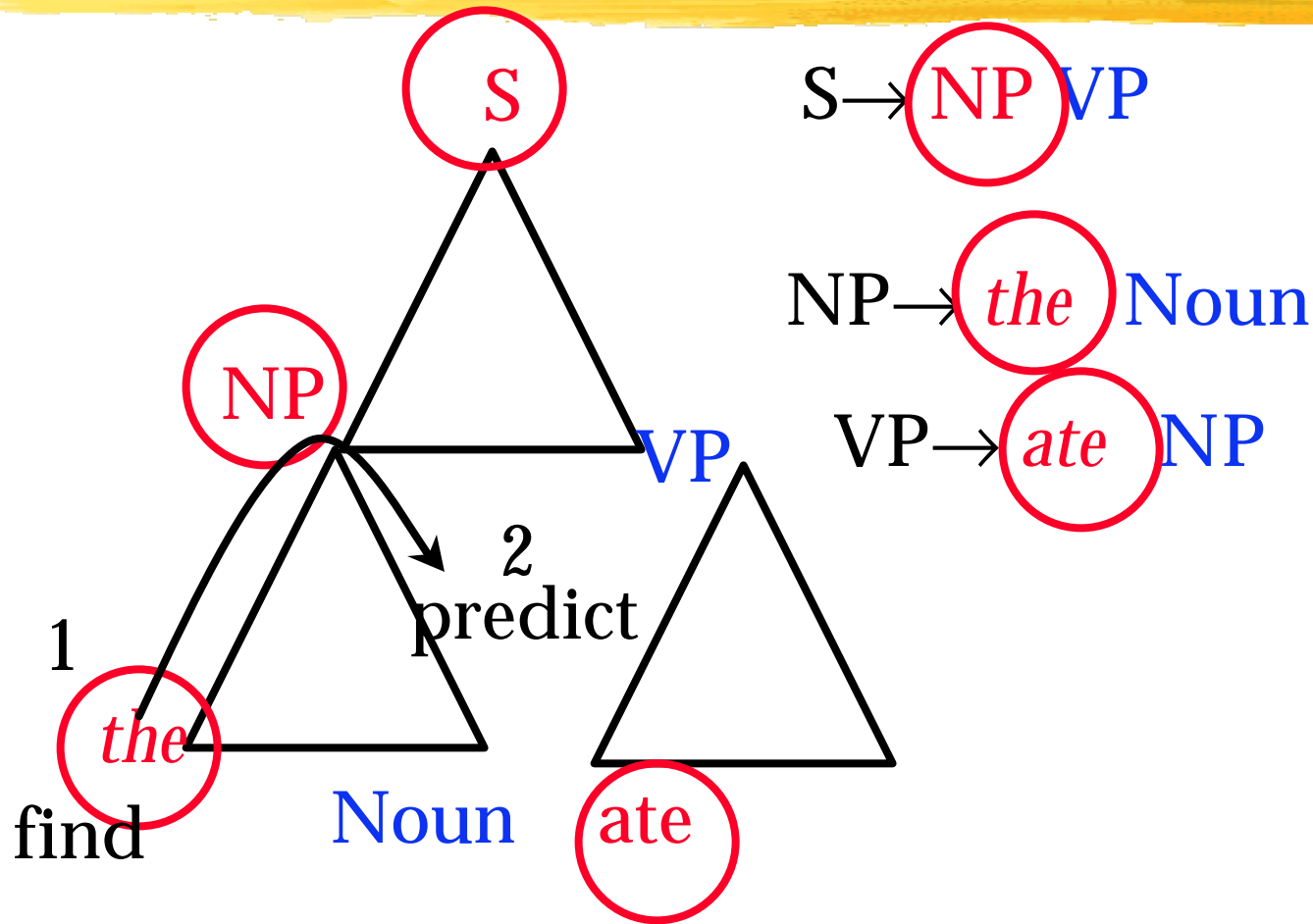
Can now 'interpret' this
pair syntactic, 'semantic' rule:

$VP \rightarrow V \ NP$, *apply* $VP(NP)$

One more search space enumeration that will be of some value

- Left-corner parsing
- Looks bottom-up in serial fashion for the *first* symbol (left-corner) of a phrase; and then tries to *confirm* the rest of the phrase top-down
- Tries to combine best features of b-u and t-d
- Clearly geared to the way a particular language (eg English) is set up

A picture of left-corner parsing



This works well



- In a language like English:
- A head-first language (function-argument)
- What about German, Dutch, Japanese?
- *dat het meisje van Holland houdt*
- *“that the girl from Holland liked”*
- These are head-final languages



What about constructing
grammars?