6.863J Natural Language Processing
Lecture 10: Charting a course through hierarchical parsing

Instructor: Robert C. Berwick
berwick@ai.mit.edu

The Menu Bar

- Administrivia:
  - Lab 2a/2b due today; 3a out Weds, due next Weds; 3b Friday – due after vacation

Agenda:

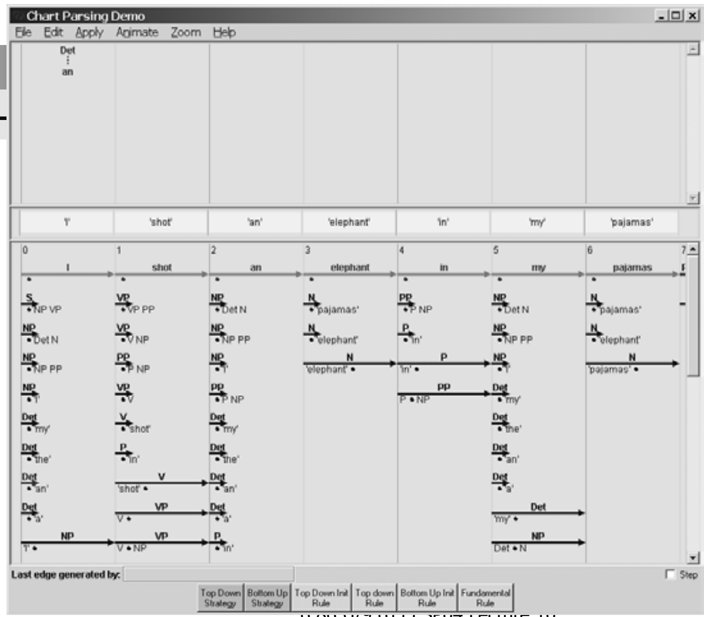
Parsing strategies: chart parsing as all-purpose search data structure – algorithm & time complexity;

CKY and Earley algorithm

What do people do?

- Preview of Lab 3

The Chart



Why a Chart?

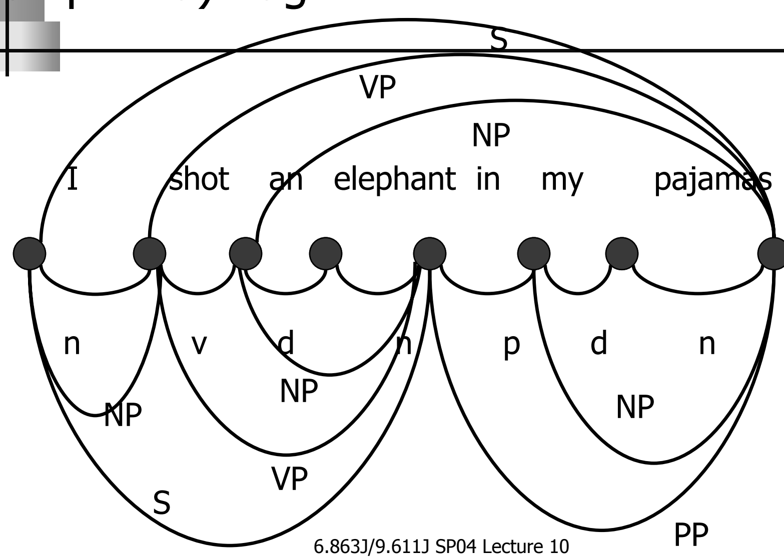
Does this ever happen in natural languages?

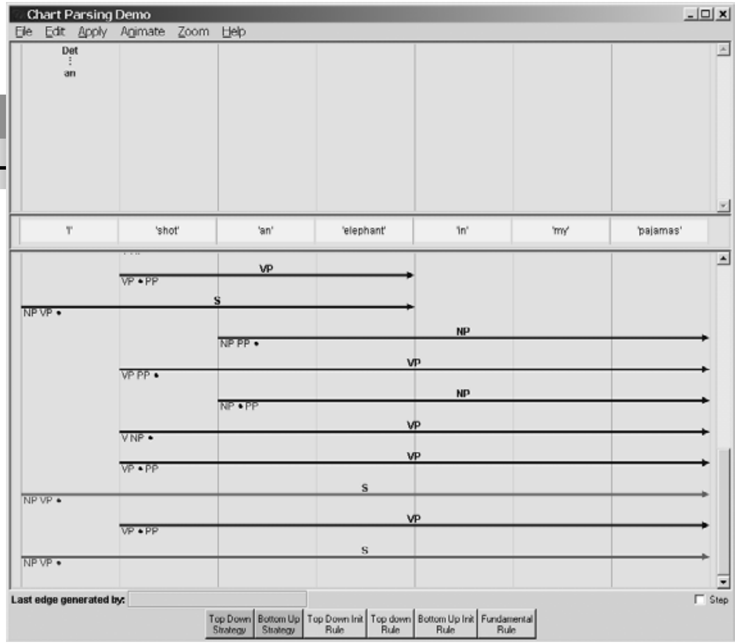
- It does if you write cookbooks… this from an actual example (from 30M word corpus)

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 $6^{\#\text{conjuncts}}$)

Chart we displayed has only *inactive* (completed) edges



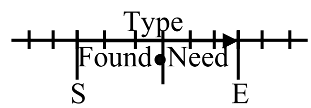


Summary so far…

- Chart: Set of edges (arcs), = state of nondeterministic automaton at step i
- Each edge characterizes a completed or partial constituent spanning a group of words
- Edge is: Dotted PhraseName[start, stop]
- Active edge: edge which still has words/phrases to be found
- Inactive edge: completed phrase
- Two operations <u>on</u> edges: ‘blow-up’, and ‘boil down’ (aka ‘paste together’); blow-up has 2 subparts: t-d or b-u

Edges (continued)

S →NP •VP [0, 2]



- An edge consists of:
  - S: A start index (1...n)
  - E: An end index (1...n)
  - Type: A phrase type (NP, PP, etc.)
  - Found: What we've found so far (list of phrase types)
  - Need: What we still need (list of phrase types)

Chart parsing

- Chart data structure + <u>Agenda</u> (a queue that will maintain set of edges to work on)
- How we decide to work on the Agenda determines the type of strategy

6.863J/9.611J SP04 Lecture 10

Chart parsing algorithm, take 1

- Initialize and invoke RULE to add active edges to AGENDA

  Until AGENDA is empty

  For each EDGE in AGENDA

  Add EDGE to chart and

  Apply RULE to EDGE, adding any new edges to AGENDA

  Apply fundamental rule to EDGE, adding any new edges to AGENDA

  Report any complete parses in AGENDA

Chart parsing strategies

- Chart, edge operations, plus:
- Ordering of how to apply the rules, and pull edges from the agenda queue

Chart Parser Rules: only 3!

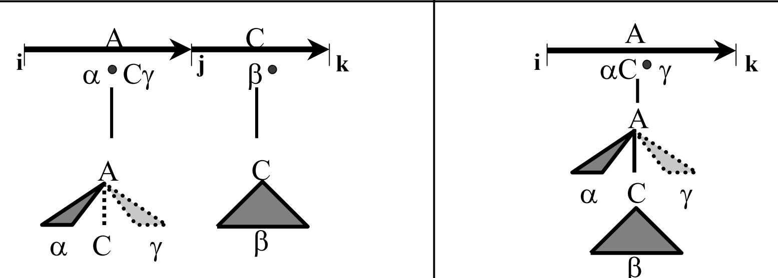
- A chart parser *rule* adds new edges to the chart.
- Each chart parsing *strategy* defines a set of rules and how they are applied
  - Top down:
    - top-down initialization rule
    - top-down rule
    - fundamental rule
  - Bottom-up:
    - bottom-up rule
    - fundamental rule

Other strategies possible -

The Fundamental Rule (AKA "paste")

- The fundamental rule is used by both top-down and bottom-up strategies.

If the chart contains: | Then add:



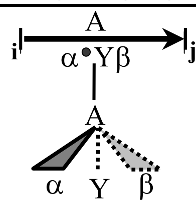
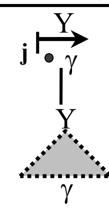
Rule a: Top-Down Rule (“TD Predict”)

- Top-down initialization:

  For any rule $S \rightarrow \alpha$:

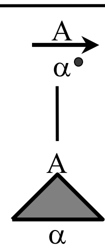
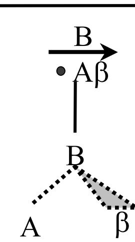
  - Add $S \rightarrow \bullet\ \alpha$ to the left side of the chart (start = end = 0).

- Top-down rule (expansion)

If the chart contains:	For each rule:	Add:
	$Y \rightarrow \gamma$	

Rule b: Bottom-Up Rule (“BU Predict”)

- Bottom-Up Rule

If the chart contains:	For each rule:	Add:
	$B \rightarrow A\beta$	

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

  $S_i \leftarrow \varnothing$ 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

Overall algorithm simple – t-d strategy

Apply top-down initialization rule – fill in pos from words (from tagger or kimmo)

- Apply top-down expansion rule to make new edges, until closure
- Apply fundamental rule (slinky extension) to make new rules until closure
- If no more active edges, stop
- Otherwise, loop to step 2

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

Top-down initialization

```
# for each production in the grammar:
>>> for production ingrammar.productions():
# does the production expand the start-symbol of the grammar?
...
      if production.lhs() == grammar.start():
...            loc =chart.location().start_location()
...            chart.insert(self_loop_edge(production,
                                           location))
```

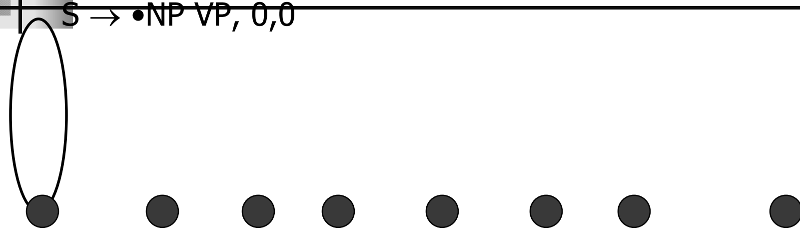
Self-loop= a zero-width edge

Overall algorithm

➨Apply top-down initialization rule – fill in pos from words (from tagger or kimmo)
- Apply top-down expansion rule to make new edges, until closure
- Apply fundamental rule (slinky extension) to make new rules until closure
- If no more active edges, stop
- Otherwise, loop to step 1

Example: Top-down init w/ chart

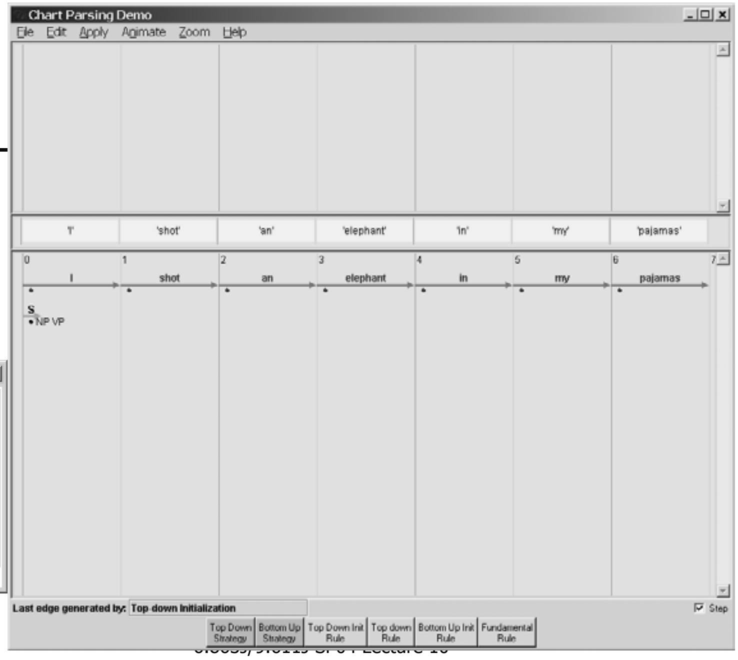
$S \rightarrow \bullet NP\ VP,\ 0,0$

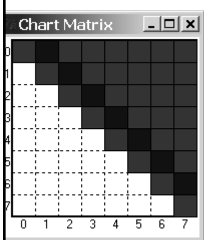


0 I 1 2 shot 3 an 4 elephant 5 in 6 my 7 pajamas

We are constructing
State set S_0 -

Initial





Overall algorithm

Apply top-down initialization rule – fill in pos from words (from tagger or kimmo)

➨ Apply top-down expansion rule to make new edges, until closure

2. Apply fundamental rule (slinky extension) to make new rules until closure
3. If no more active edges, stop
4. Otherwise, loop to step 2

Operation 2: Top down edge creation

- If 'dot' is before a phrase, we want to predict or wish for it
- Example: $S \rightarrow \bullet NP\ VP$
- Means we should look for an NP next
- So we should add all the possible ways to find an NP
- This means self-loops, all starting at this position, labeled $NP \rightarrow \bullet$ etc…[0,0]
- We do this until we get terminal elements

Python

```
# for each production in the grammar:
  >>> for production in grammar.productions():
# for each incomplete edge in the chart:
...   for edge in chart.incomplete_edges():
# does the expected constituent match the production?
 ...     if edge.next() == production.lhs():
...          location = edge.location().end_location()
...            chart.insert(self_loop_edge(production,
                                           location))
```

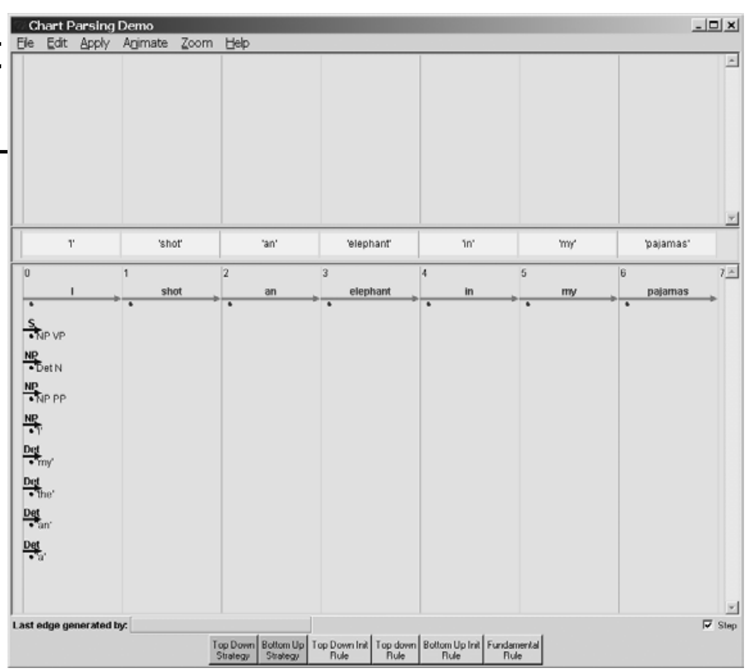
In picture form

S → •NP VP
NP → • D N (from td rule, or predict)
NP → • I
NP → • NP PP
+ all POS expansions

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

State set S_0 now done

Chart



Overall algorithm

Apply top-down initialization rule – fill in pos from words (from tagger or kimmo)

- Apply top-down expansion rule to make new edges, until closure

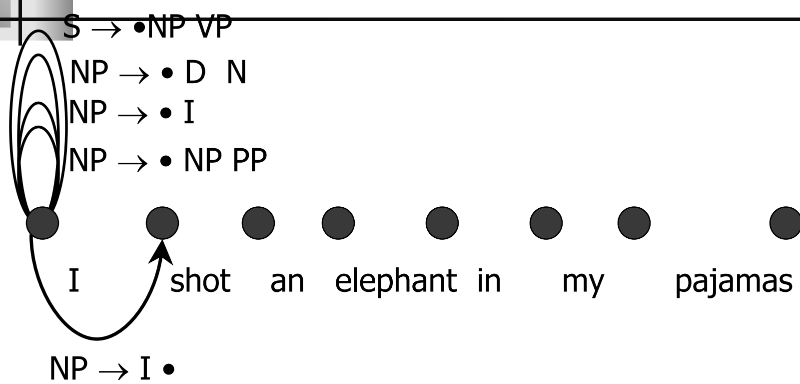
➨ Apply fundamental rule (slinky extension) to make new rules until closure

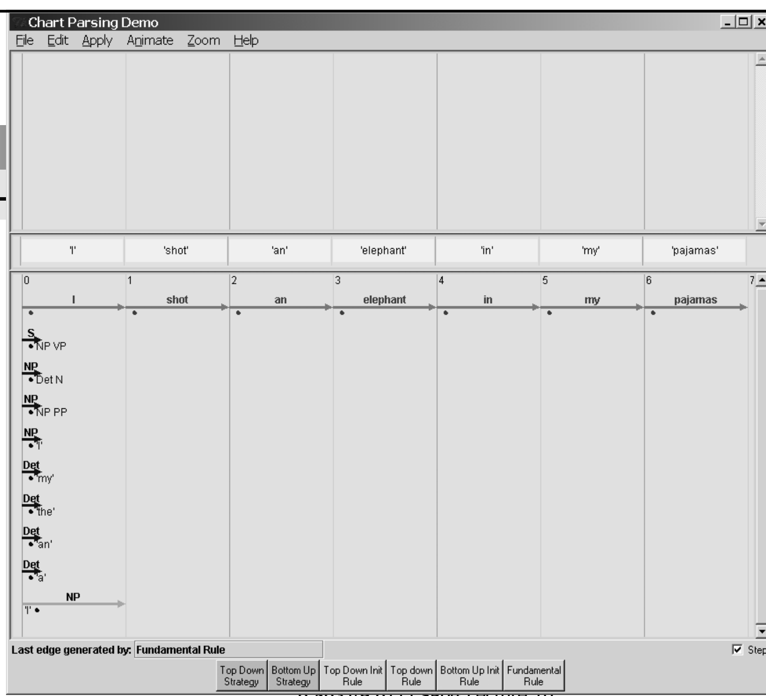
- If no more active edges, stop
- Otherwise, loop to step 2

Edge extension: Fundamental Rule

- Applies whenever we can <u>extend</u> the RHS of a phrase
- Two places: (1) Dot is before an element, and that element is in the input ('scan'); (2) dot is at end of the rhs of a rule
- We have found the phrase's longest right-hand extent (and we know where the phrase started)
- Means the word or phrase is complete, and we have <u>confirmed</u> the lhs of the rule
- In this case: NP now extended from 0,0 to 0,1

Scan (fundamental rule) to next word…follow the bouncing dot…



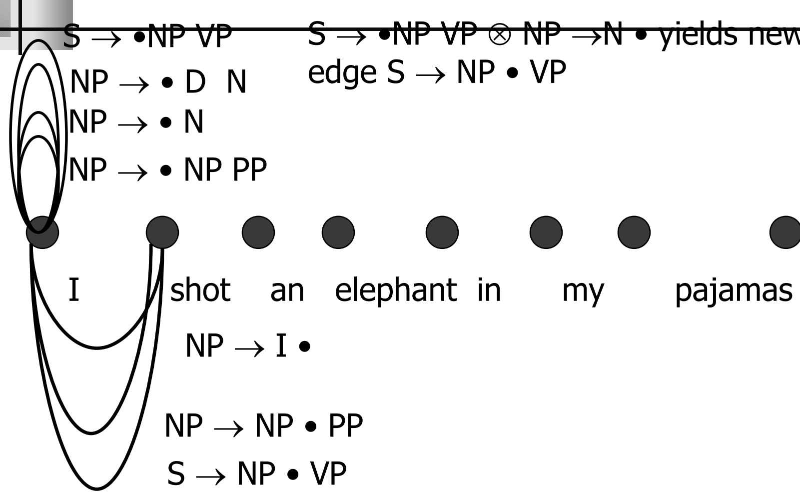


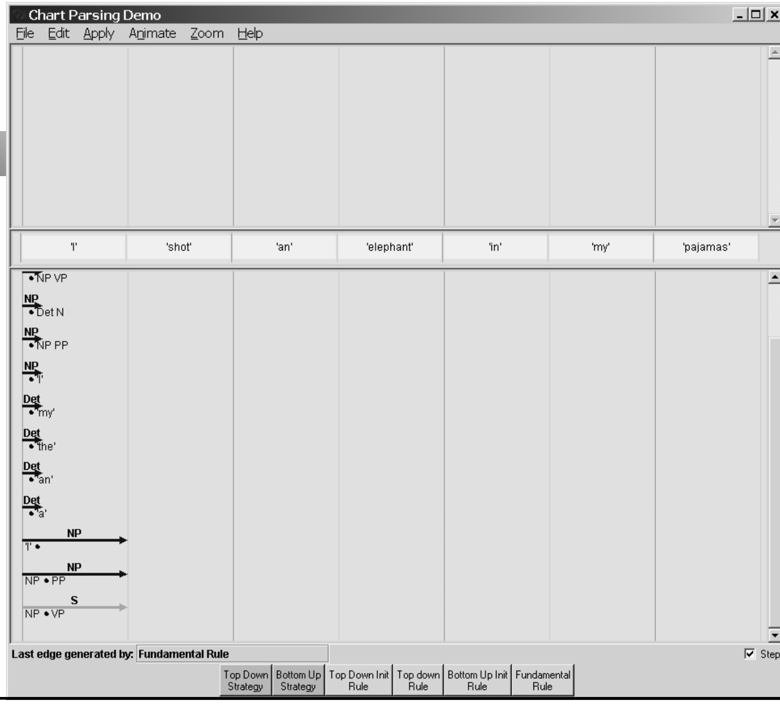
The Fundamental Rule Applies…

- As time goes by…
- Actually, as NP goes by…
- We can also extend the length of all the <u>other</u> edges that had an NP with a dot before them…
- That is,

Dot at end…so we 'complete' NP

S → •NP VP ⊗ NP →N • yields new edge S → NP • VP

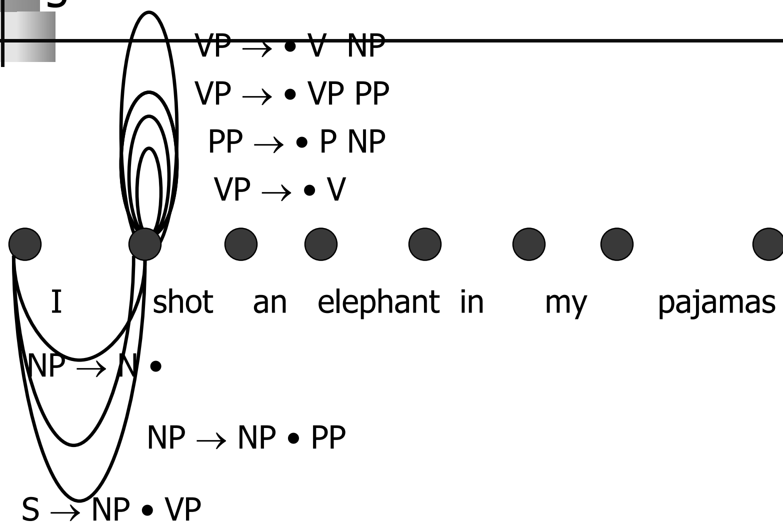


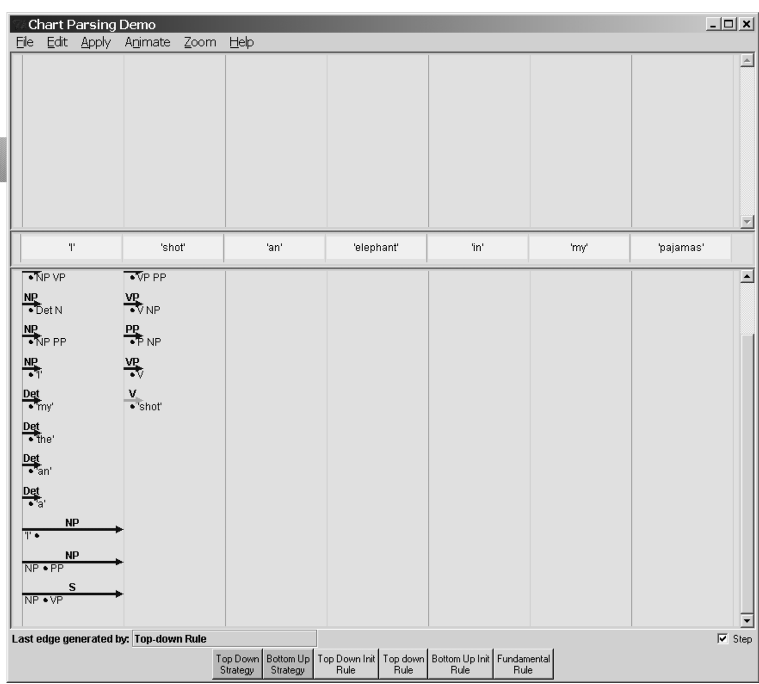


What next?

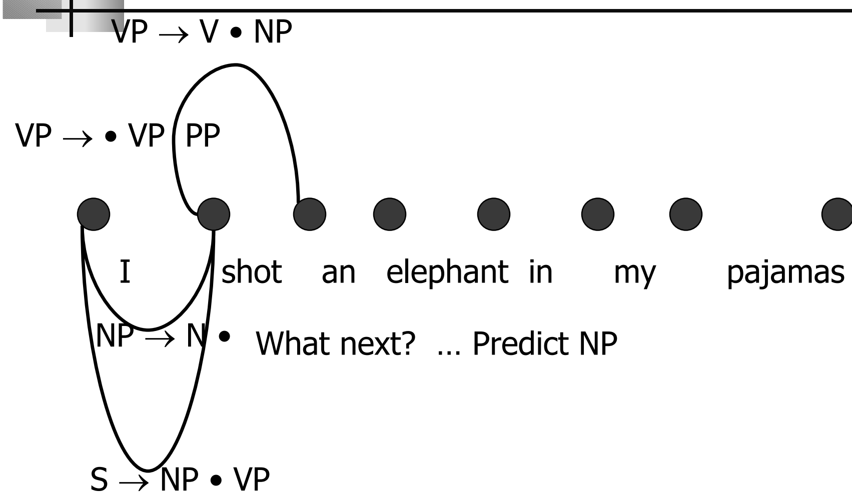
- ➼Apply fundamental rule until closure
- Then Top-down rule again, until closure

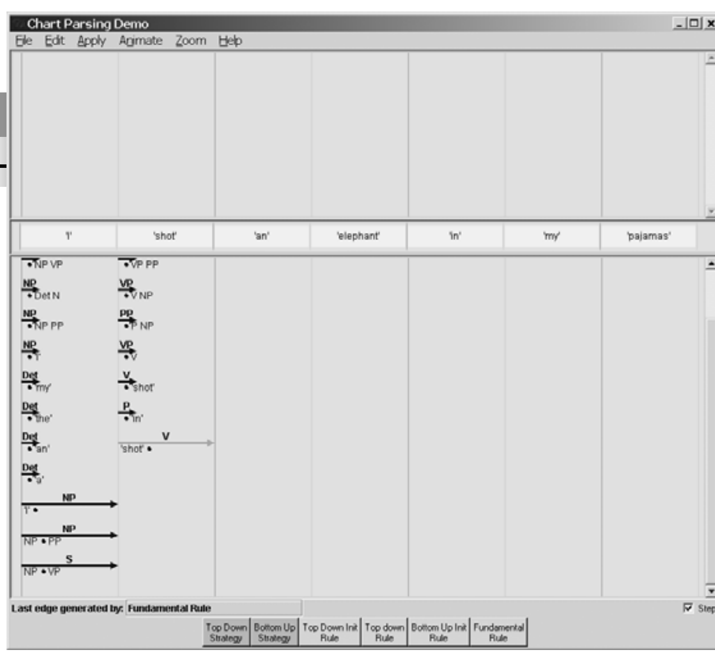
Loop: And now top-down expansion again

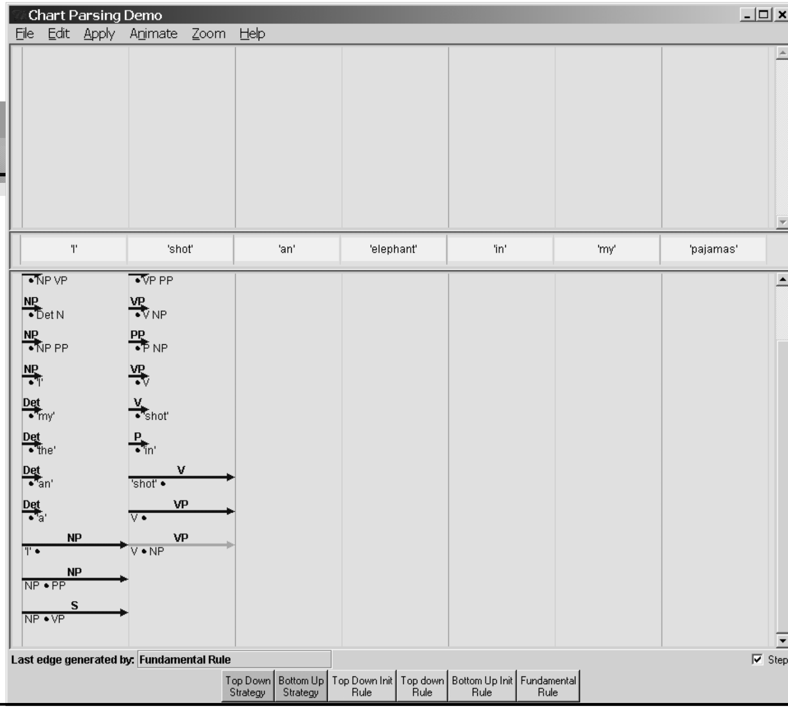




Scan Verb, via Fundamental rule







So this strategy is:

- Apply top-down init rule
- Apply top-down rule, until closure
- Apply fundamental rule, until closure
- Go back, loop, until no more rules apply

The ops add edges in our full chart representation …

1. [Top-down edge processor]: Loops (Predict) – start a phrase: top-down
2. [Word edge processor]: Skips (Scan) – build phrase from word – aka Fundamental Rule applied to one word or POS
3. [Fundamental Rule]: Pastes – glue 2 edges to make a third, larger one (Complete) – finish a phrase (the Fundamental rule)

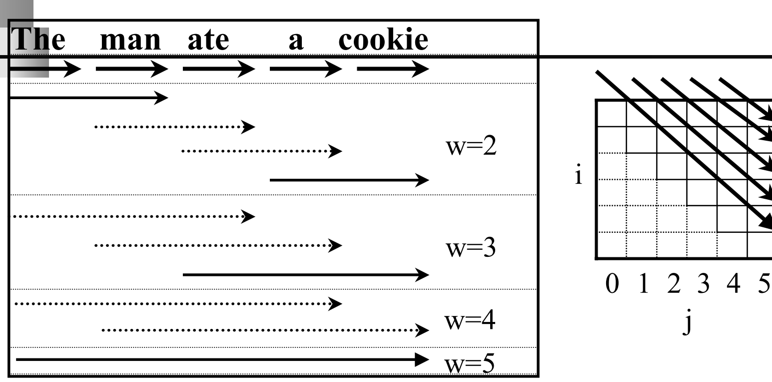
Charting a course

- Many different strategies possible
- Let's see pure bu
- then top-down w/ bottom up filtering = Earley parser
- Point: <u>many</u> difft ways of 'filling in' the chart (but not <u>all</u> possible ways!) – correspond to "coherent" ways to explore the phrase search space

CKY (Cocke-Kasami-Younger)

- A bottom-up chart parsing strategy
- Requires a grammar in Chomsky Normal Form
  - Binary branching nonterminal rules
    - $A \rightarrow B\ C$
  - Unary terminal rules
    - $A \rightarrow w$
- First, add lexical edges for each word.
- Then, for each width w (2 to N):
  - Scan left to right, combining edges to form new edges with width w

CKY: Overview



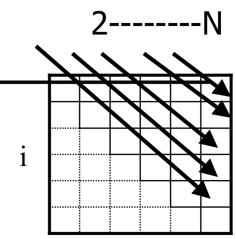
- First, add the lexical edges
- Then, for each w, add edges of length w

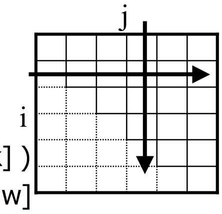
CKY: Algorithm

- First, add the lexical edges
- Then:

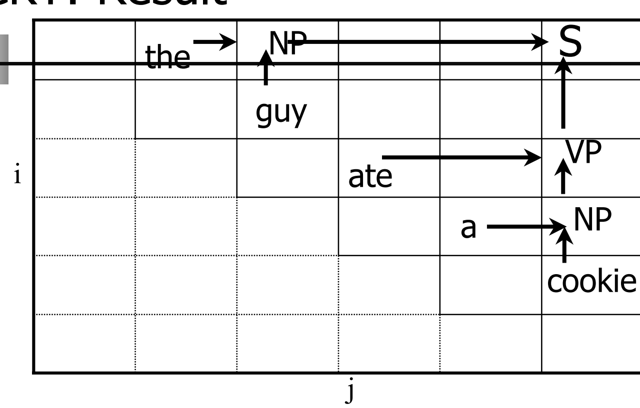
```
for  w = 2  to  N:
   for  i = 0 to N-w:
            for  k = 0  to  w-1:
                     If ( A→BC and
                               B→α ∈ chart[i,k] and
                               C→β ∈ chart[i+k,i+w-k] )
                               Add A→BC to chart[i,i+w]
```

- If S∈chart[0,N], return the corresponding parse



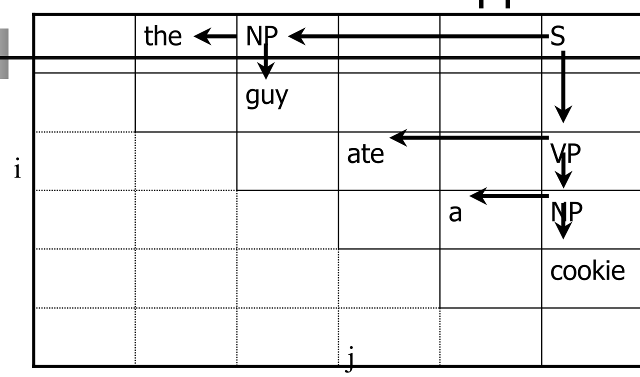


CKY: Result



- Use backpointers to remember what we combined

The fundamental rule applies…



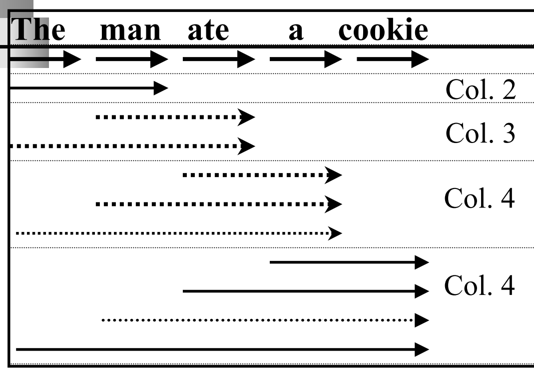
- Use backpointers to remember what we combined

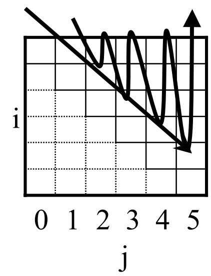
Chart as a Matrix

- We can represent a chart as an upper triangular matrix.
- chart[i,j] is the set of dotted rules that span [i:j]

i \ j	0	1	2	3
0	S → • NP VP NP → • John	John → • NP → John • S → NP • VP		S -> NP VP •
1		VP → • V NP V → • saw	saw → • V → • saw VP → V • NP	VP → V NP •
2			NP → • Mary	Mary → • NP → Mary•
3				

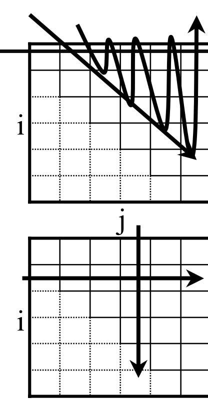
Left-to-Right BU: Overview





- First, add the lexical edges
- Then scan left to right, combining edges

Left-to-Right BU: Algorithm



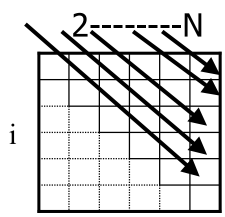
- First, add the lexical edges
- Then:

```
for  j = 1  to  N:
   for  i = 0  to  N-j:
              for  k = 1  to  j-2:
                        If ( A→BC and
                                  B→α ∈ chart[i,k] and
                                  C→β ∈ chart[i+k,j-k] )
                                  Add A→BC to chart[i,j]
```

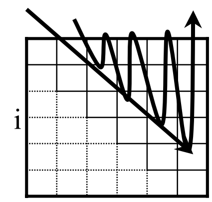
- If $S \in$ chart[0,N], return the corresponding parse

Comparison of construction patterns

for w = 2 to N:
  for i = 0 to N-w:
    for k = 0 to w-1:

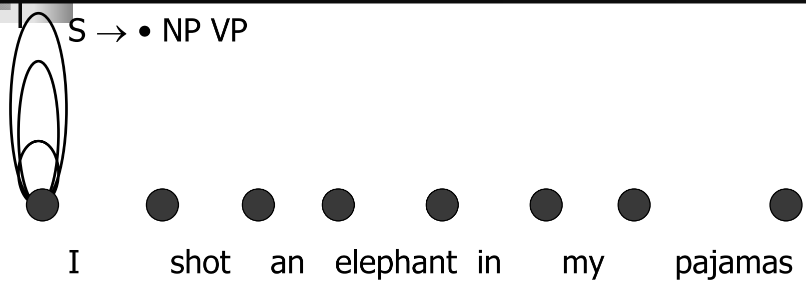


for j = 1 to N:
  for i = 0 to N-j:
    for k = 1 to j-2:

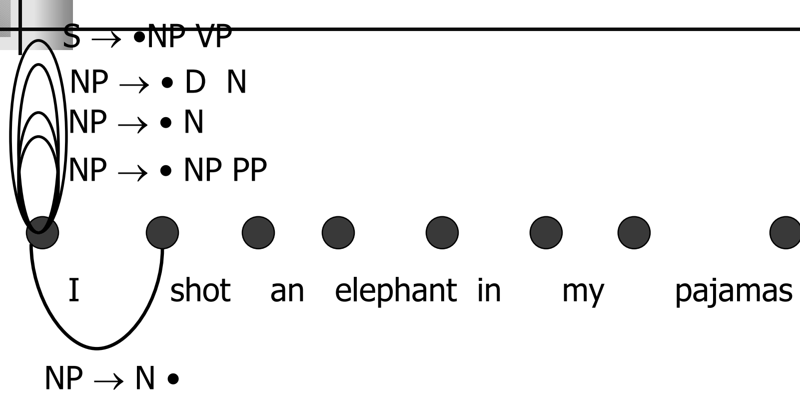


	FTN Parser	CFG Parser
Initialize:	Compute initial state set S_0 1. $S_0 \leftarrow q_0$ 2. $S_0 \leftarrow$ eta-closure (S_0) q_0= [Start→•S, 0] eta-closure= transitive closure of jump arcs	Compute initial state set S_0 1. $S_0 \leftarrow q_0$ 2. $S_0 \leftarrow$ eta-closure (S_0) q_0= [Start→•S, 0, 0] eta-closure= transitive closure of Predict and Complete
Loop:	Compute S_i from S_{i-1} For each word, wi, 1=1,...,n $S_i \leftarrow \cup_{q \in S_{i-1}} \delta(q, w_i)$ $S_i \leftarrow$ e-closure(S_i)	Compute S_i from S_{i-1} For each word, w_i, 1=1,...,n Si$\leftarrow \cup_{q \in S_{i-1}} \delta(q, w_i)$ = Scan(S_{i-1}) q=item $S_i \leftarrow$ e-closure(S_i) e-closure= closure(Predict, Complete)
Final:	Accept/reject: If $q_f \in S_n$ then accept; else reject q_f= [Start→S•, 0]	Accept/reject: If $q_f \in S_n$ then accept; else reject q_f= [Start→S•, 0, n]

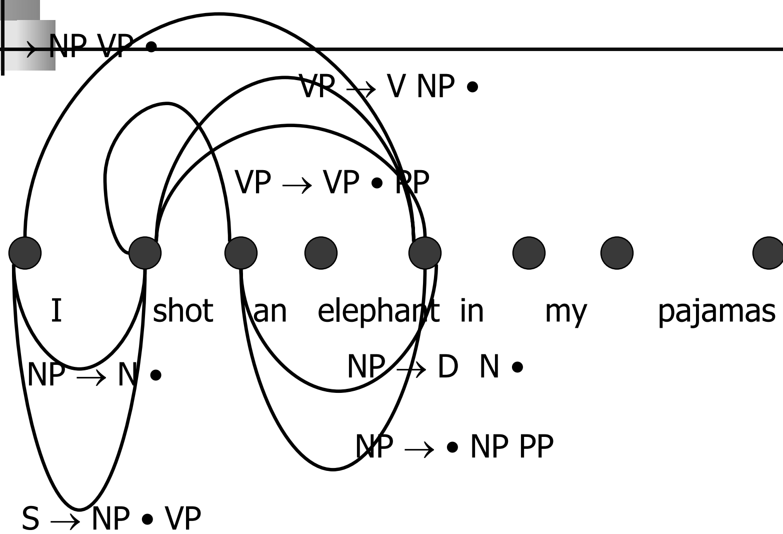
Picture: <u>Predict</u> (Top-down rule) adds the 'loops'



Picture: <u>Scan</u> (Fundamental rule) adds the 'jumps'



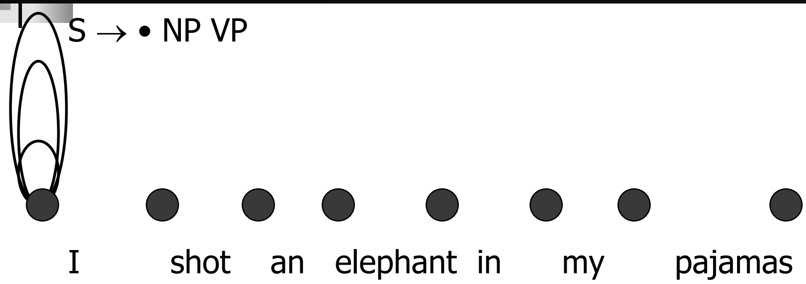
Picture: Complete (Fundamental rule) combines edges



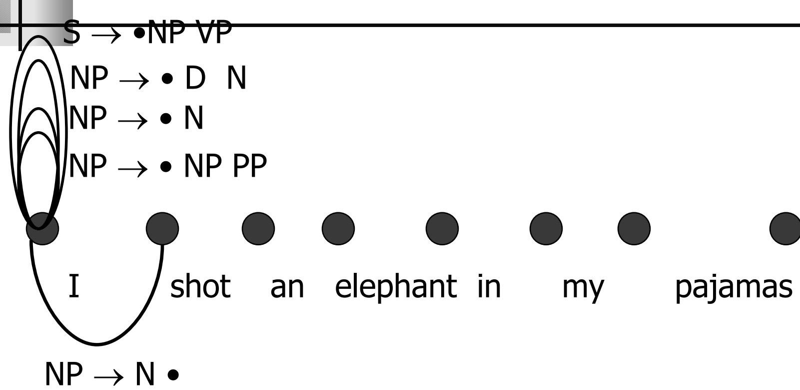
Earley's Algorithm

- Top-down chart parsing strategy
  - With bottom-up filtering
- Applicable with any Grammar
- First, <u>initialize</u> with the top-down init rule:
  For every grammar rule $S \rightarrow \alpha$:
  Add $\xrightarrow[\bullet\alpha]{S}$
- Then, go left to right, applying 3 rules:
  - Predictor (=top-down rule)
  - Scanner (=fundamental rule on terminals)
  - Completer (=fundamental rule on nonterminals)

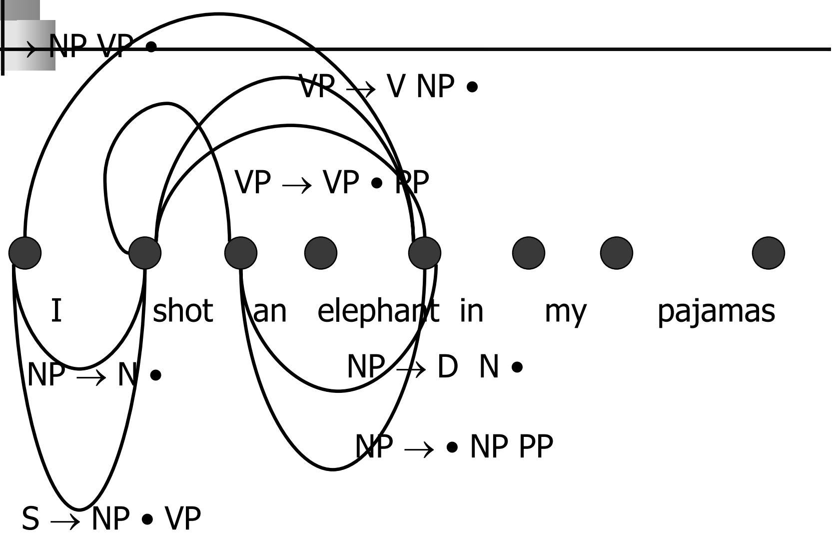
Picture: <u>Predict</u> adds the ‘loops’



Picture: <u>Scan</u> adds the ‘jumps’



Picture: <u>Complete</u> combines edges (The "fundamental rule")



The ops

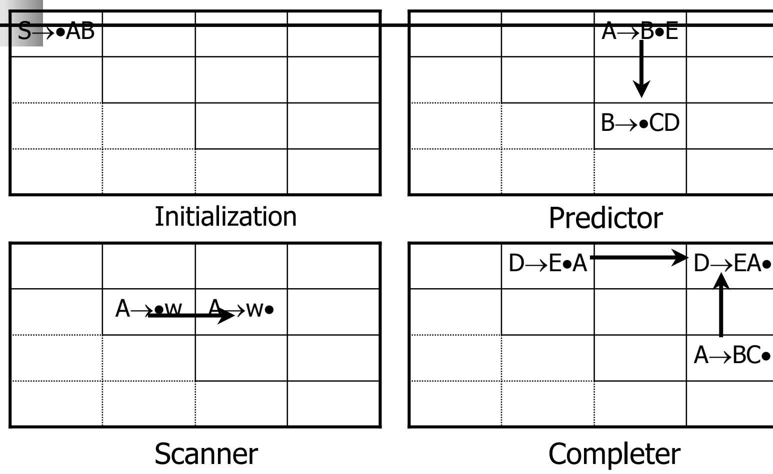
- 3 ops: *scan, predict, complete;* or
  scan, push, pop

1. *Scan:* <u>move</u> forward, consuming a token (word class) - what if this is a *phrase name*, though?
2. *Predict (push):* <u>start</u> building a phrase (tree) at this point in the input; or jump to subnetwork;
3. *Complete (pop):* <u>finish</u> 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

Earley’s Algorithm: Rules



Earley’s Algorithm

- For each column (j) (= State Set) maintain a queue of edges.
- Initialization:
  - For every grammar rule S→α, add $\xrightarrow[\bullet\alpha]{S}$ to queue[0]
- Process queues from left to right (0 to N).
  - For each edge in the queue, apply one of 3 rules:
    - If it’s incomplete, and the next symbol after the dot is a preterminal (i.e., a part of speech tag), apply scanner.
    - If it’s incomplete, and the next symbol after the dot is not a preterminal, apply predictor.
    - If it’s complete, apply completer.

Earley's Algorithm: Main

For each rule $S \rightarrow \alpha$ in the grammar:
  Add $S \rightarrow \bullet\alpha$ to chart[0,0]

For i = 0 to N:
  for edge in queue[i]:

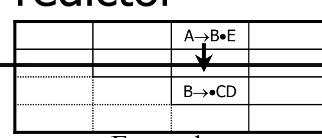
    if edge is incomplete and edge.next is a part of speech:
      scanner(edge)

    if edge is incomplete and edge.next is not a POS:
      predictor(edge)

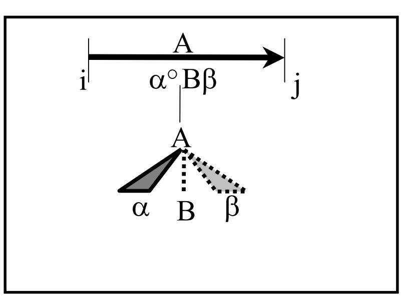
    if edge is complete:
      completer(edge)

Earley's Algorithm: Predictor

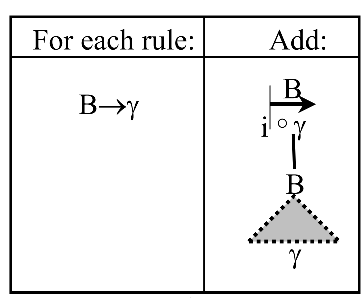
- Predictor($A \rightarrow \alpha \bullet B\beta$, [i,j])



Example



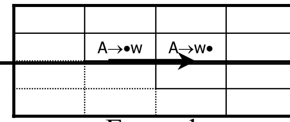
Input



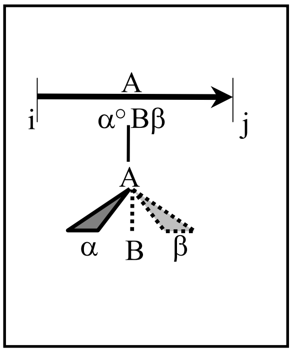
Rule

Earley's Algorithm: Scanner

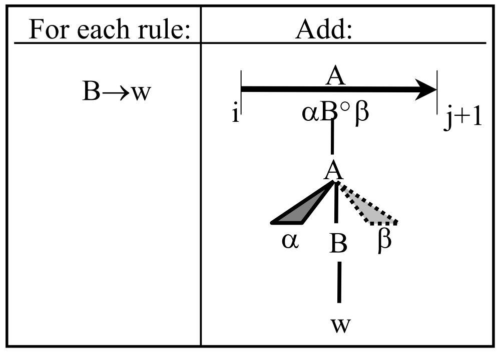
- Scanner($A \to \alpha \bullet B\beta$, [i,j])



Example



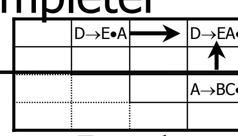
Input



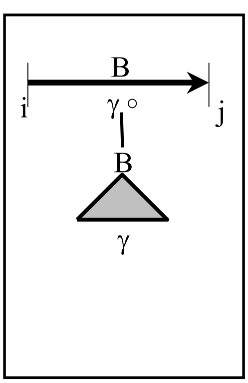
Rule

Earley's Algorithm: Completer

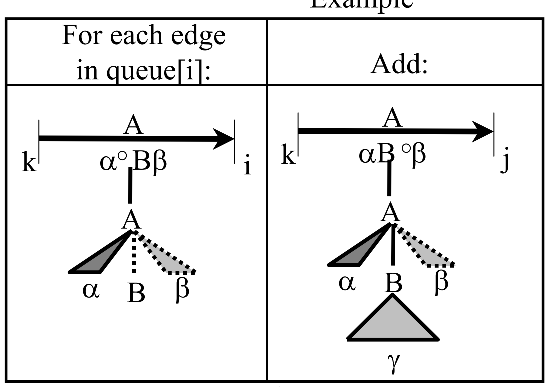
- Completer($B \to \gamma \bullet$, [i,j])



Example



Input



Rule

Definitions – words & symbols

- <u>Scan</u>

  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

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

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…

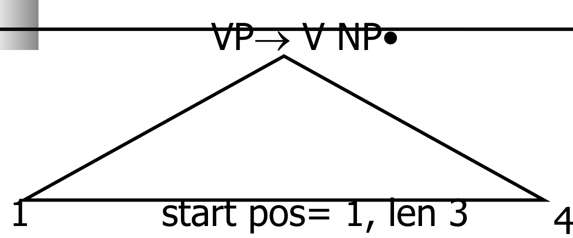
- <u>Predict (Push):</u>
  - Before: $[A \rightarrow \alpha \bullet B\ \beta,\ k,\ i]$, B=nonterminal, in S_i then
  - After: <u>Add</u> all new edges of form $[B \rightarrow \bullet\ \gamma,\ i+1,\ i+1]$ to State Set S_{i+1}

Complete (finish phrase)

- Suppose current edge e is finished (dot at rh end). Suppose e looks like:

  $X \rightarrow y_1 \;\; y_2 \;\; \ldots \; y_p \;\bullet$ from start pos k, length m
- <u>Check</u> 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

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



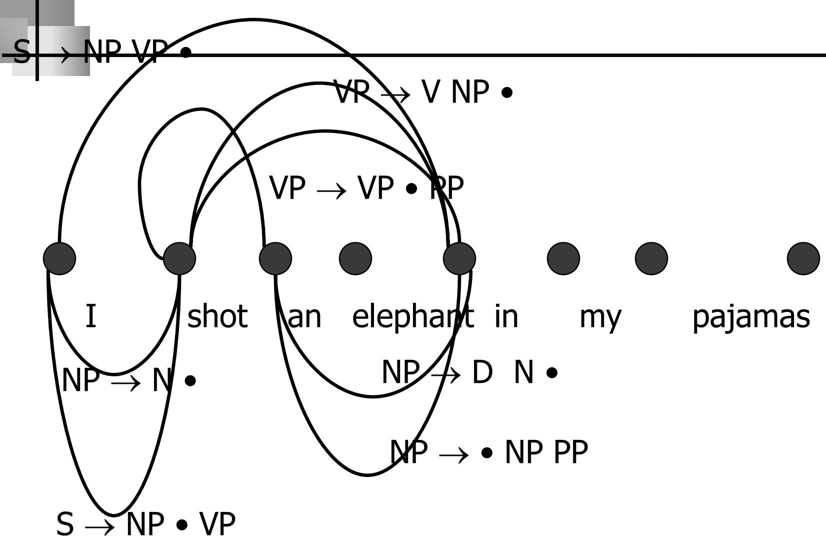
Go back to state set 1 & see what rule was looking for a VP

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

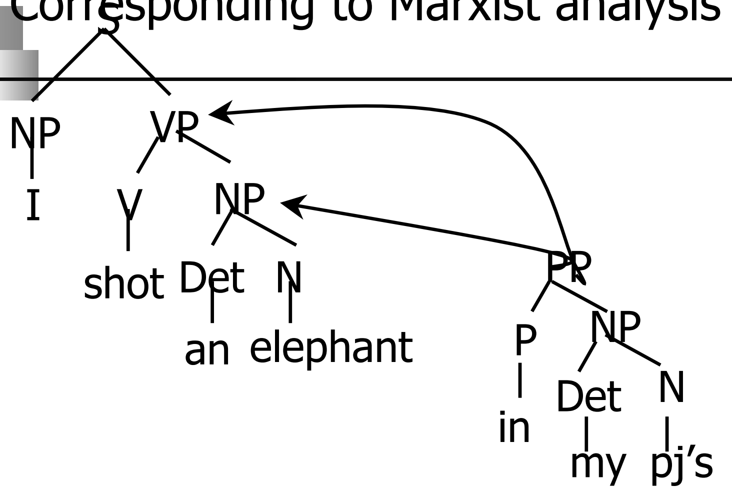
More precisely

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

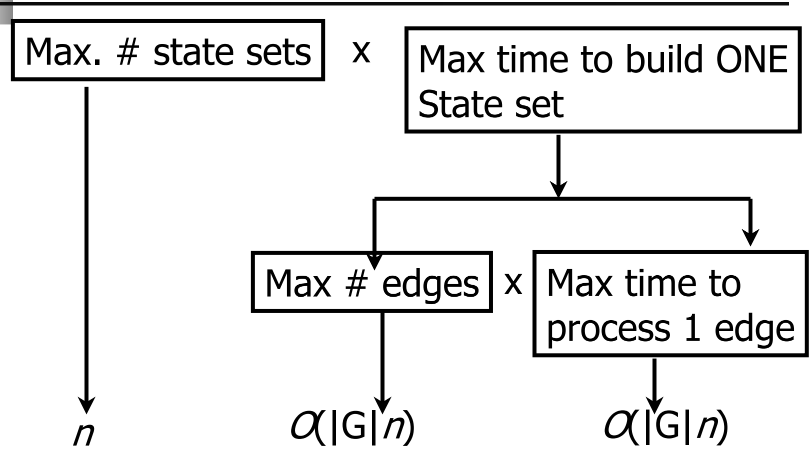
Picture: Complete <u>combines</u> edges



Corresponding to Marxist analysis



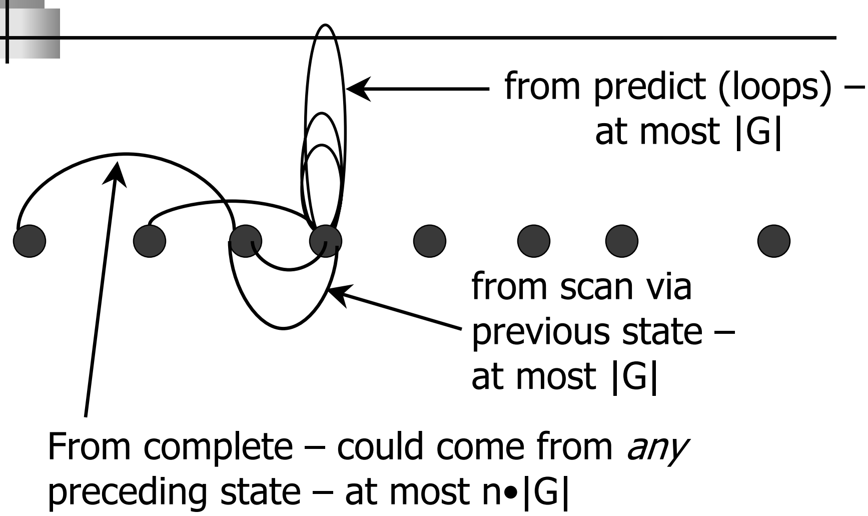
So time complexity picture looks like this:



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., $|G|$x n
  - (Why is this?)
  - (Edges have form: dotted rule, start, len)
- Note use of grammar size here: amount of ‘chalk’ = Σ # symbols in G.

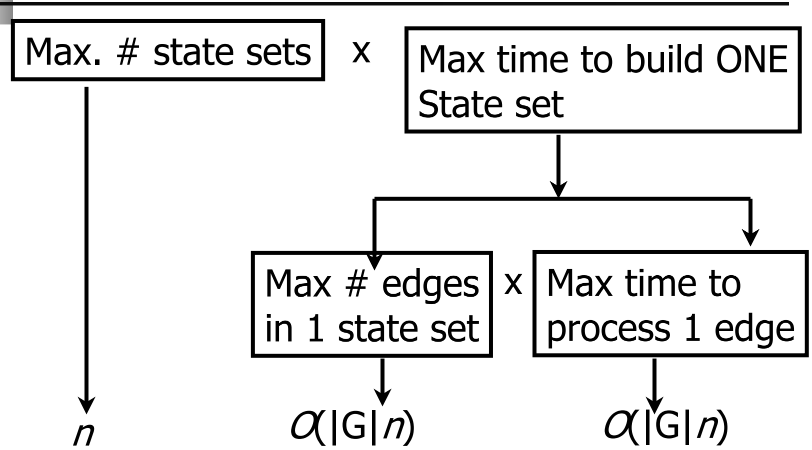
Max # distinct edges: loops, incoming from scans, incoming from paste:



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 <u>*complete*</u> !
- For this, we have to go back to previous state set and look at <u>*all*</u> (in worst case) edges in <u>that</u> state set - and we just saw that in the worst case this could be O (|G|x 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