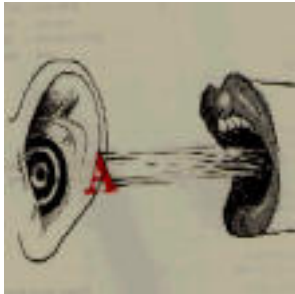


# Lecture 7: Context-free parsing & statistical context-free parsing



Professor Robert C. Berwick  
berwick@csail.mit.edu

# The Menu Bar

- Administrivia: RR#2 out today
- Efficient context-free parsing: Three Loops to Rule them All, Three Loops to Bind them
- Removing redundancy redundancy
- Statistical context-free parsing
- Parsing tricks; the Earley algorithm

# Review: Context-free grammar (CFG) More precisely

A *context-free grammar* (CFG) is a 4-tuple  $(N, \Sigma, P, S)$  where:

$N$  is a finite set of nonterminal symbols (phrase names, categories);

$\Sigma$  is a finite set of terminal symbols ('lexical items', 'parts of speech', eg, noun, verb, auxiliary verb, ...);

$P$  is a set of production rules  $\langle A \in N, \alpha \rangle$ , where  $\alpha$  is a sequence of terminal or nonterminals; sometimes also written in the form

$X \rightarrow Y_1 Y_2 \dots Y_n$  for  $n \geq 0$ ,  $X \in N$ ,  $Y_i \in (N \cup \Sigma)$

$S \in N$  is a designated start symbol.

We write the productions as  $A \rightarrow \alpha$  ('is-a')

Note: this is already off the mark from human language in at least two ways! (What ways?)

# Languages, grammars, and all that

- A (terminal) string  $s \in \Sigma^*$  is in the language generated by the context-free grammar  $G$  iff there is at least one derivation that yields  $s$
- The set of strings derivable (in the language of)  $G$  is denoted  $L(G)$ , the grammar's *weak generative capacity*
- A string may have more than one derivation (as in the fsa case!) = *ambiguity*
- *Parsing* = given an input sentence  $w$ , CFG  $G$ , recover the (set of derivations) of  $w$  wrt  $G$
- And that is what makes parsing hard! Why?

# Canonical derivations in CFGs

- **Left-most derivation** in a CFG:

Sequence of derivation lines  $s_1, s_2, \dots, s_n$

where  $s_1 =$  Start symbol

$s_n \in \Sigma^*$ , that is, all terminal symbols

For  $i=2, \dots, n$ , each  $s_i$  is derived from  $s_{i-1}$  by selecting the leftmost nonterminal  $X$  in  $s_{i-1}$  and replacing it by some  $\beta$ , the right-hand side of a rule  $X \rightarrow \beta$  in  $P$

(Can you tell me how to map derivations to ‘hierarchical structures’ – no, no, do not use the ‘T’ word....)

# CFGs, languages, and derivations

- The derives relation  $\Rightarrow$
- Define wrt grammar  $G = (N, \Sigma, P, S)$  as follows  
 $\alpha \Rightarrow \beta$  iff  $\exists \alpha_1, \alpha_2$  s.t.  $\alpha = \alpha_1 A \alpha_2$ ;  $\beta = \alpha_1 \gamma \alpha_2$ ; and  
 $A \rightarrow \gamma \in P$ . (Some rule rewrites  $\alpha$  as  $\beta$ )
- Reflexive, transitive closure of  $\Rightarrow$  is  $\Rightarrow^*$

If  $\alpha, \beta$  is in  $\Rightarrow^*$  then we say that  $\alpha$  derives  $\beta$  (by 0 or more steps)

The *language generated* by a cfg  $G$  is the set of terminal strings  $w \in \Sigma^*$  s.t.  $S \Rightarrow^* w$  (also called the yield of  $S$ )

# Derives relation & added info in context-free grammar

- Relates all elts by *either* dominance or precedence
- Induces a (derivation) tree (Q: do we lose any information in this tree?)
- FSA represents pure *linear* relation: what can *precede* or (*follow*) what
- CFG adds a single new predicate: *dominate*
- Claim: The dominance and precedence relations amongst the words exhaustively describe its *syntactic* structure
- When we parse, we are recovering these predicates

# Derivation induces 2 relations, dominance & precedence

- Binary Relation  $D$ , dominance:

$A D v$  iff  $\exists \alpha_1, \alpha_2$  s.t.  $\alpha \Rightarrow \beta$  via  $A \rightarrow \alpha_1 v \alpha_2$

- Binary relation  $<$  precedence:

$v < w$  iff  $\exists \alpha_1, \alpha_2$  s.t.  $\alpha = \alpha_1 v w \alpha_2$  or  $\beta = \alpha_1 v w \alpha_2$  &  $\alpha \Rightarrow \beta$

Confirm that our derivation steps previously induce such relations... note that all elts are related by  $<$  or  $D$ .

(Suppose not...?)

# A context-free grammar

$S \rightarrow NP V$

$NP \rightarrow Det N$

$NP \rightarrow NP PP$

$VP \rightarrow VP PP$

$VP \rightarrow V NP$

$PP \rightarrow P NP$

$Det \rightarrow a$

$S \rightarrow NP VP$

$N \rightarrow caviar$

$N \rightarrow spoon$

$V \rightarrow spoon$

$V \rightarrow ate$

$P \rightarrow with$

$Det \rightarrow the \mid a$

$N \rightarrow guy$

A derivation of:  
*Madoff ate caviar with a spoon*

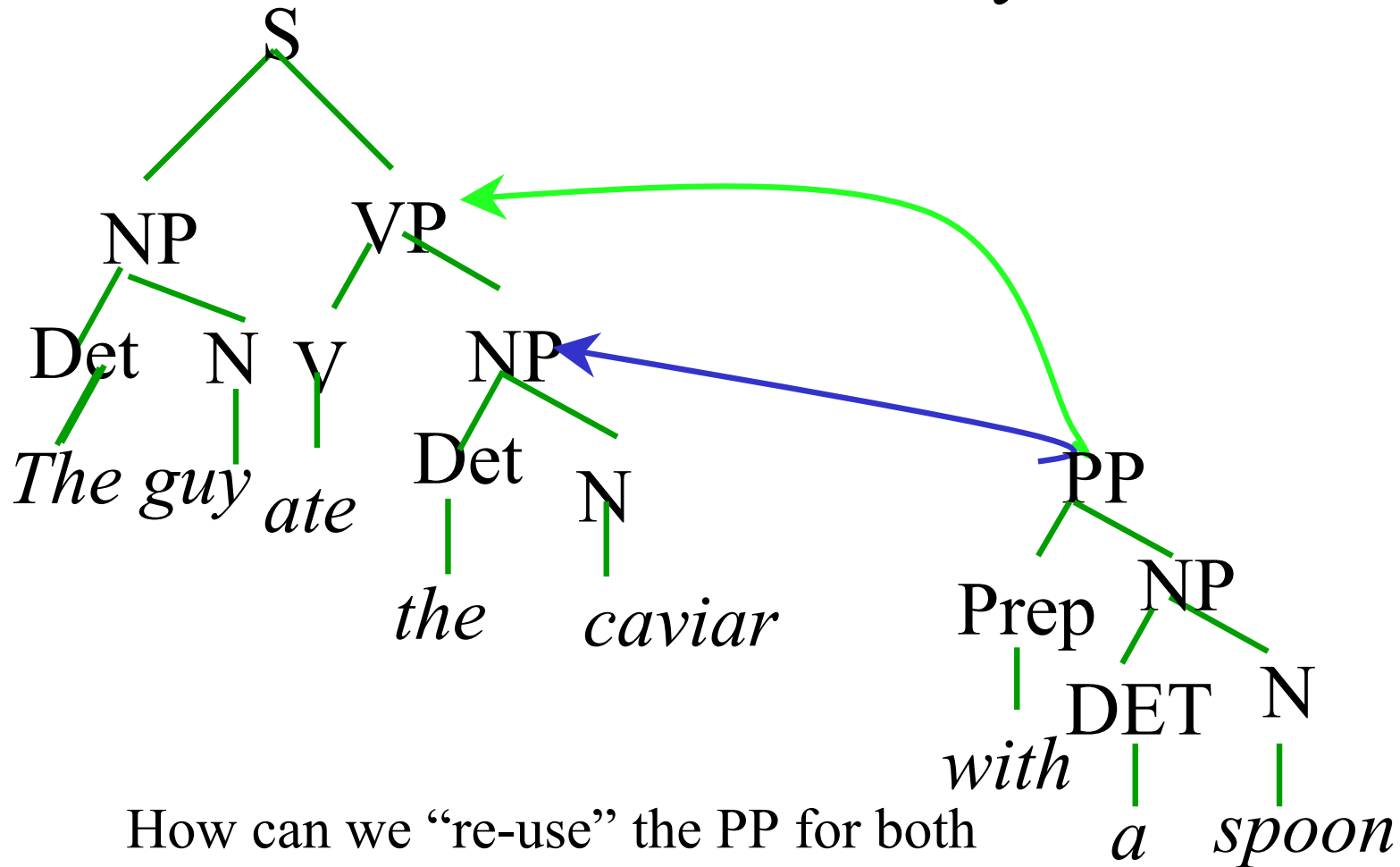
- |                                 |  |
|---------------------------------|--|
| 1. Root                         | 11. NP V Det N <i>with a spoon</i>                       |
| 2. S                            | 12. NP V Det <i>caviar with a spoon</i>                  |
| 3. NP VP                        | 13. NP V <i>the caviar with a spoon</i>                  |
| 4. NP V NP                      | 14. NP <i>ate the caviar with a spoon</i>                |
| 5. NP V NP PP                   | 15. Det N <i>ate the caviar with a spoon</i>             |
| 6. NP V NP P NP                 | 16. Det <i>guy ate the caviar with a</i><br><i>spoon</i> |
| 7. NP V NP P Det N              | 17. The guy ate the caviar with a                        |
| 8. NP V NP P Det <i>with</i>    | spoon  |
| 9. NP V NP P <i>a spoon</i>     |  |
| 10. NP V NP <i>with a spoon</i> |  |

Now parse this...

# Some simple methods

- Shift-reduce parsing: stack + two operations....*shift* item onto stack, or *reduce* (via grammar rule); bottom up
- What's the problem with this?
- Recursive descent parsing: top-down

# Marxist analysis



# Local & Global ambiguity makes parsing hard

- Local ambiguity

- Plural/singular: *the sheep are/is...*
- *The chair is too wobbly for the woman to sit on/on it*

- Global ambiguity

NP → NP PP | PP → P NP, *the guy on the hill with the telescope*

<i>n</i>	1	2	3	4	5	6	7	8
#	2	5	14	132	469	1430	4862	16796

Noun compounds: NP → NN NN, *water meter cover screw*

Conjunctions: NP → NP *and* NP

NP → NP *and* NP *and* NP

<i>n</i>	1	2	3	4	5	6	7	8	9
#	1	1	3	11	45	197	903	4279	20793

# Why is parsing hard?

## Martha Stewart's revenge

- If you write cookbooks...this from an actual example in a 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...*

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

Use dynamic programming (aka  
'memoization' to avoid building same phrase  
more than once  
(actually: A\* search through phrase space...  
when we add some notion of "distance"))

Let's see how...

# A context-free grammar

$S \rightarrow NP VP$

$NP \rightarrow N$

$S \rightarrow NP V$

$N \rightarrow \textit{guy} \mid \textit{Papa}$

$NP \rightarrow \textit{Det} N$

$N \rightarrow \textit{caviar}$

$NP \rightarrow NP PP$

$N \rightarrow \textit{spoon}$

$VP \rightarrow VP PP$

$V \rightarrow \textit{spoon}$

$VP \rightarrow V NP$

$V \rightarrow \textit{ate}$

$PP \rightarrow P NP$

$P \rightarrow \textit{with}$

$\textit{Det} \rightarrow \textit{a}$

$\textit{Det} \rightarrow \textit{the}$

(suppose VP rule was  $VP \rightarrow V NP..$  What then?)

Tabular, bottom-up parsing:  
CKY, recognition version  
Uses a 'table', aka a 'chart'

**Grammar:** binary branching form, rules  $N_k \rightarrow N_l N_m$

**Input:** string of  $n$  words

**Output:** yes/no;  $table(i, j)$  of phrases spanning  $i, j$ ,  $split(i, j, k)$  of backptrs

**Data structure:**  $n \times n$  table

rows labeled 1 to  $n$  (indexed by  $j$ )

columns labeled 1 to  $n$  (indexed by  $i$ )

$table(i, j)$  lists constituents found covering words  $i$  thru  $j$

# CKY algorithm

Initialize:

For  $j = 1, \dots, n, k = 1, \dots, K$

$table(j, j) \leftarrow \{N_k | N_k \rightarrow w_j\}$

For  $j = 1$  to  $n$

For  $i = j - 1$  downto  $1$

For  $s \leftarrow i$  to  $j$

For  $N_k \rightarrow N_l N_m$  s.t.  $N_{l,m} \in G$  &

$N_l \in table(i, s)$  &

$N_m \in table(s + 1, j)$

$table(i, j) \leftarrow table(i, j) \cup N_k$

$split(i, j, k) \leftarrow \{s, l, m\}$

*the guy ate the caviar with a spoon*



<i>i j</i>	1	2	3	4	5	6	7	8
1	1,1							
2								
3								
4								
5								
6								
7								
8								

# The CKY algorithm: 3 Loops to Rule Them All

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

```

for j ← from 1 to n (# words) do
    table[j, j] ← {A | A → words[j] ∈ G}
    for i ← from j-1 downto 1 do
        for s ← i to j do
            table[i, j] ← table[i, j] ∪
                {Nk | Nk → NlNm ∈ G,
                 Nl ∈ table[i, s],
                 Nm ∈ table[s+1, j]}
            split(i, j, k) ← {s, l, m}
    
```

← loop 1: across cols, to fill in p.o.s.  
← loop 2: look up row by row  
← loop 3: find all possible ways A can be pasted together looking lft across row and down col

Note: We will have to augment to this to store backptrs in table to recover *parses*

*j=1 to n left to right across columns*

*i=j downto 1 ; up the rows*

*s=i to j ; look left & down*

Step 1: fill table(0,1) [off-diagonal]

*j=1, table[j,j]=table[1,1]=POS 'the'= Det*

*the guy ate the caviar with a spoon*

Column

<i>i j</i>	1	2	3	4	5	6	7	8
1	Det <sub>[1,1]</sub>							
2								
3								
4								
5								
6								
7								
8								



*j=1 to n left to right across columns*

*i=j-1 downto 1; up the rows*

*s=i to j; look left & down*

Step 3:  $j=3$ ,  $table[j-1, j]=table[3,3]='ate' = V$

$i=j-1=2$  to 1 ; try to fill 2 cells above [3,3]

$i=2$  ; try to fill  $table[2,3]$

$s=i=1=2$  to  $j=3$

$table[2,3]=table[2, 2] \otimes table[3,3] =??$

$i=1$  ; try to fill  $table[1,3]$

$s= 1$  to  $j=3$  ; note 2 splits  $s$

$table[1,3]=table[1, 1] \otimes table[2,3] =??$

$table[1,3]=table[1, 2] \otimes table[3,3] =S$

*the guy ate the caviar with a spoon*

<i>i j</i>	1	2	3	4	5	6	7	8
1	Det <sub>[1,1]</sub>	NP <sub>[1,2]</sub>	NP <sub>[1,3]</sub>					
2		N <sub>[2,2]</sub>						
3			V <sub>[3,3]</sub>					
4								
5								
6								
7								
8								

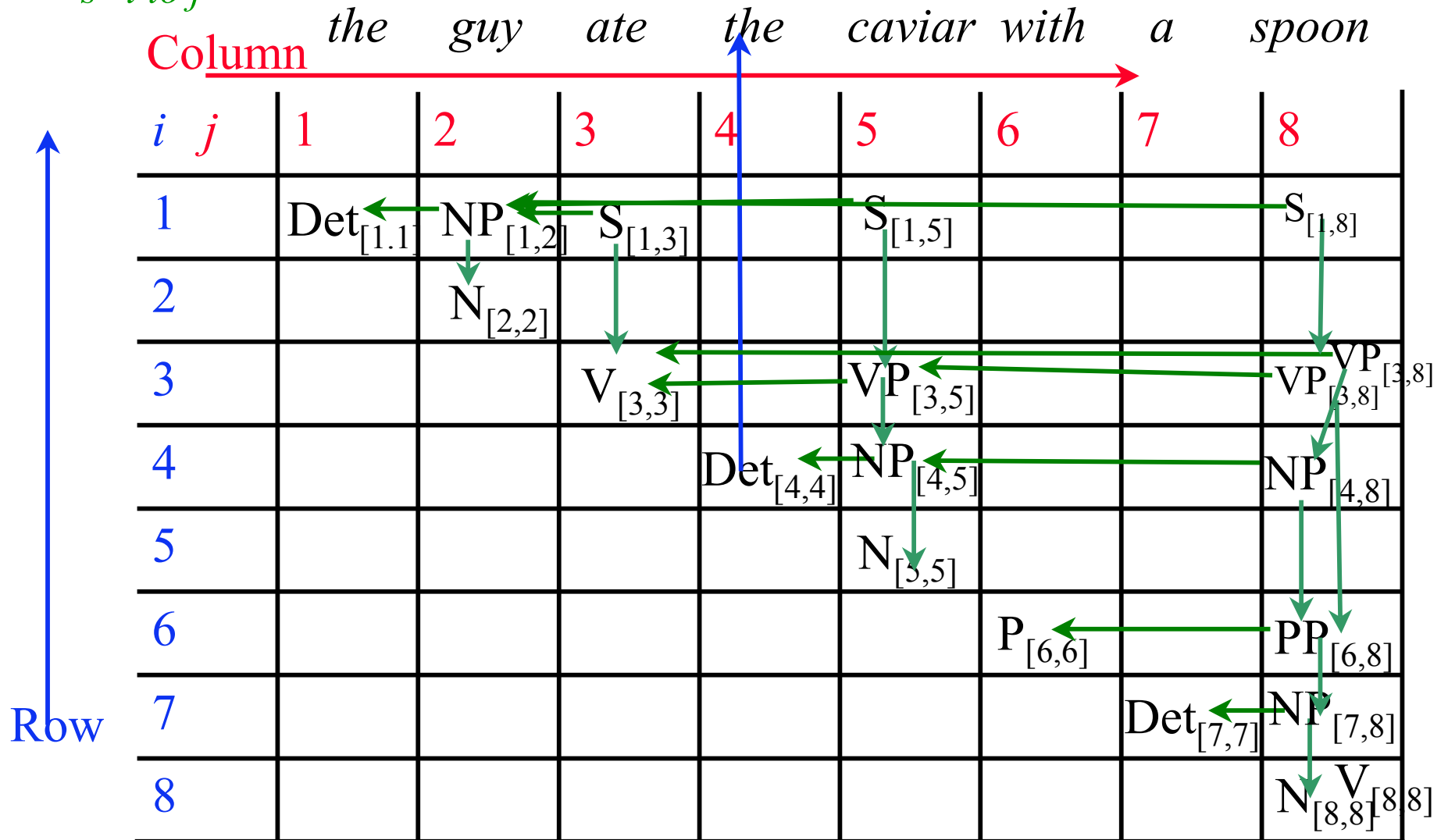
6.863J/9.611J SP11

*j=1 to n left to right across columns*

*i=j-1 downto 1; up the rows*

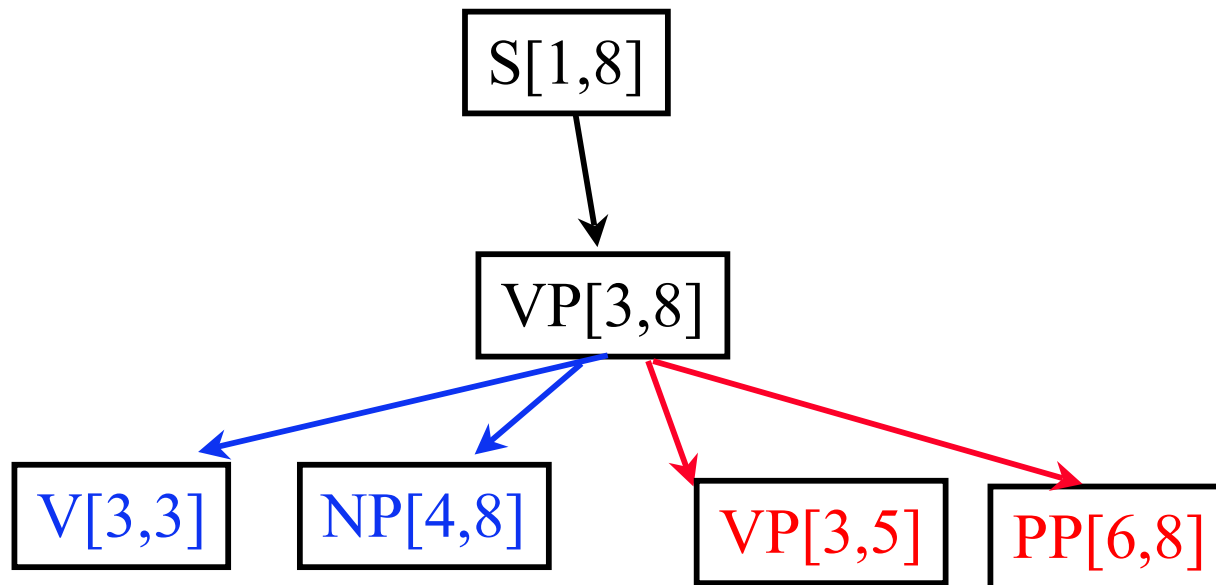
*s=i to j*

*j=8*



Note how ambiguity is ‘packed’ into the the *split* pointers!

- Two VP[3,8] entries - are they distinct?
- Yes! Via backpointers. This adds time!



Need to add pointers: ‘which rule caused me to be added?’

# Example from textbook

*Book the flight through Houston*

S, VP, Verb, Nominal, Noun [0,1]				S <sub>1</sub> , VP, X <sub>2</sub>
		S, VP, X <sub>2</sub> [0,3]		S <sub>2</sub> , VP
				S <sub>3</sub>
	Det [1,2]	NP [1,3]		NP [1,5]
		Nominal, Noun [2,3]		Nominal [2,5]
			Prep [3,4]	PP [3,5]
				NP, Proper- Noun [4,5]

# Time complexity

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

**for**  $j \leftarrow$  **from** 1 to  $n$  (# words) **do**

$table[j, j] \leftarrow \{A \mid A \rightarrow words[j] \in G\}$

**for**  $i \leftarrow$  **from**  $j-1$  **downto** 1 **do**

**for**  $s \leftarrow i$  **to**  $j$  **do**

$table[i, j] \leftarrow table[i, j] \cup$

$\{N_k \mid N_k \rightarrow N_l N_m \in G,$

3 loops of  $O(n)$ , where  $n$  is the sentence length  $N_l \in table[i, s],$

$N_m \in table[s+1, j]\}$

# Time complexity & grammar size

- Inner loop: how many times do we execute:  
**for**  $s \leftarrow i$  **to**  $j$  **do**

$$table[i,j] \leftarrow table[i,j] \cup$$

$$\{N_k \mid N_k \rightarrow N_l N_m \in G,$$

$$N_l \in table[i,s],$$

$$N_m \in table[s+1,j]\}$$

This is how many times we can scribble in one cell

At most, equal to the size of the grammar,  $|G|$

## Generalizing by using a chart

- Chart data structure = the matrix just shown
- But we can use a variety of search strategies: top-down, bottom-up, or in between, depending on the language, and what we know

# Probabilistic CKY

Modify CKY to find the greatest probability  $\pi(i, j, k)$ ,  
given nonterminals  $k = 1, \dots, K$

Sentence is  $n$  words long,  $w_1, w_2, \dots, w_n$

Spans word positions  $(i, j)$ , i.e.,  $(1, n, Start)$

Data structures:  $table(i, j)$ ,  $split(i, j, k)$ ,  $\pi(i, j, k)$

# Probabilistic CKY

Initialize:

For  $j = 1, \dots, n, k = 1, \dots, K$

$table(j, j) \leftarrow \{N_k | N_k \rightarrow w_j\}$

$\pi(i, j, k) \leftarrow p(N_k | N_k \rightarrow w_j)$

For  $j = 1$  to  $n$

For  $i = j - 1$  downto  $1$

$max \leftarrow 0$

For  $s \leftarrow i$  to  $j$

For  $N_k \rightarrow N_l N_m$  s.t.  $N_{l,m} \in G$  &

$N_l \in table(i, s)$  &

$N_m \in table(s + 1, j)$

$table(i, j) \leftarrow table(i, j) \cup N_k$

$prob \leftarrow p(N_k \rightarrow N_l N_m) \times \pi(i, s, l)$

$\times \pi(s + 1, j, m)$

if  $prob > max, max \leftarrow prob$

$split(i, j, k) \leftarrow \{s, l, m\}$

$\pi(i, j, k) = max$

## So is this OK?

- Where does CKY do extra work?
- Where is it not like what people do?
- Is it feasible in practice?

# Probabilistic CKY

Modify CKY to find the greatest probability  $\pi(i, j, k)$ ,  
given nonterminals  $k = 1, \dots, K$

Sentence is  $n$  words long,  $w_1, w_2, \dots, w_n$

Spans word positions  $(i, j)$ , i.e.,  $(1, n, Start)$

Data structures:  $table(i, j)$ ,  $split(i, j, k)$ ,  $\pi(i, j, k)$

# Probabilistic CKY

Initialize:

For  $j = 1, \dots, n, k = 1, \dots, K$   
 $table(j, j) \leftarrow \{N_k | N_k \rightarrow w_j\}$   
 $\pi(i, j, k) \leftarrow p(N_k | N_k \rightarrow w_j)$

For  $j = 1$  to  $n$

For  $i = j - 1$  downto  $1$

$max \leftarrow 0$

For  $s \leftarrow i$  to  $j$

For  $N_k \rightarrow N_l N_m$  s.t.  $N_{l,m} \in G$  &  
 $N_l \in table(i, s)$  &  
 $N_m \in table(s + 1, j)$

$table(i, j) \leftarrow table(i, j) \cup N_k$

$prob \leftarrow p(N_k \rightarrow N_l N_m) \times \pi(i, s, l)$   
 $\times \pi(s + 1, j, m)$

if  $prob > max, max \leftarrow prob$

$split(i, j, k) \leftarrow \{s, l, m\}$

$\pi(i, j, k) = max$

# Principle AWP

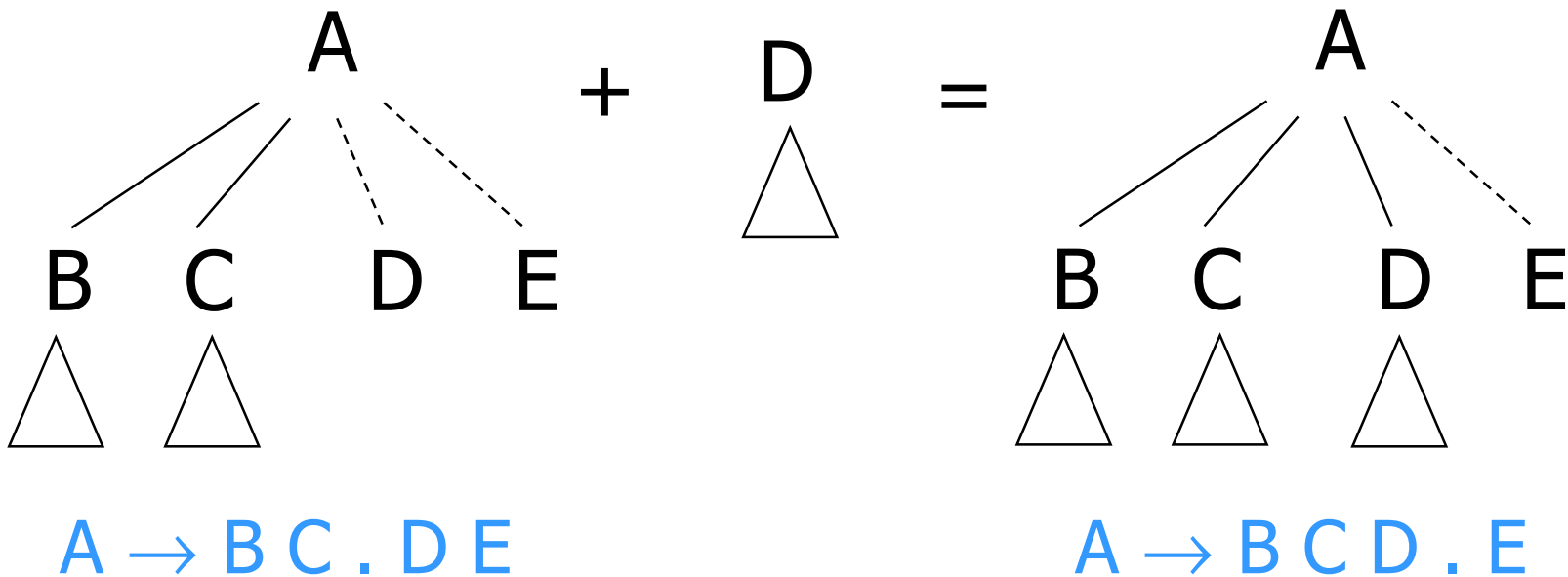
- CKY works strictly bottom-up, so it doesn't pay attention to what has happened 'on the left' beforehand - builds lots of structures that might not ever fit into the full parse
- Example: *the book John saw the guy had read*
- Fix: use left context
- Is this enough?

# Earley parsing

- Top down instead of bottom-up
- Uses left-context and optionally right-context to constrain search so we waste less time building impossible things
- No restrictions of the form of the grammar  
(e.g.,  $\text{NP} \rightarrow \text{Det N with John PP}$  is an OK rule, thanks to a clever ‘on the fly’ transformation to binary branching rules (“dotted rules”))
- Again  $O(n^3)$  time (but watch out for grammar size!)

# Overview of Earley's Algorithm

- Finds constituents and partial constituents in input
  - $A \rightarrow B C . D E$  is partial: only the first half of the  $A$



# Overview of Earley's Algorithm

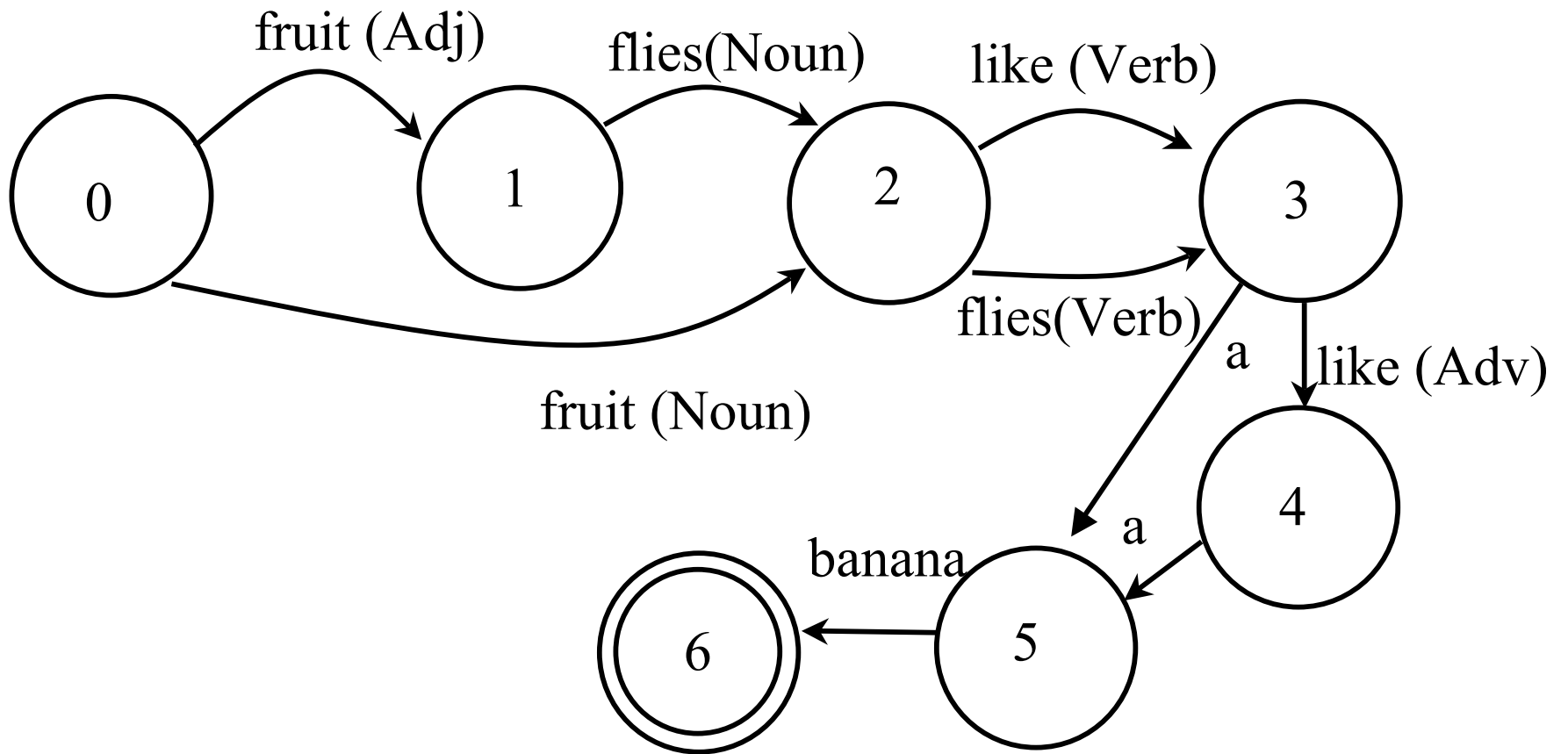
- Proceeds incrementally, left-to-right
  - Before it reads word 5, it has already built all hypotheses that are consistent with first 4 words
  - Reads word 5 & attaches it to immediately preceding hypotheses. Might yield new constituents that are then attached to hypotheses immediately preceding *them* ...
  - E.g., attaching **D** to  $A \rightarrow B C . D E$  gives  $A \rightarrow B C D . E$
  - Attaching **E** to that gives  $A \rightarrow B C D E .$
  - Now we have a complete **A** that we can attach to hypotheses immediately preceding the **A**, etc.

# Motivation

- We will carry out a *deterministic* simulation of a *nondeterministic* machine that can keep track of ‘all the possible next states’  $S_i$  it can be in after reading each word
- So we imagine a sequence of state sets,  $S_0, S_1 \dots$
- Representation as a ‘chart’ of ‘columns’; Column  $i$  or *chart*( $i$ ) will contain *all of the possible* machine states after reading the  $i^{th}$  word (NB. Indexing: we start at 0, *before* the first word of the sentence)
- Compare this to how we can solve ‘fruit flies like a banana’ in a nondeterministic finite-state automaton...

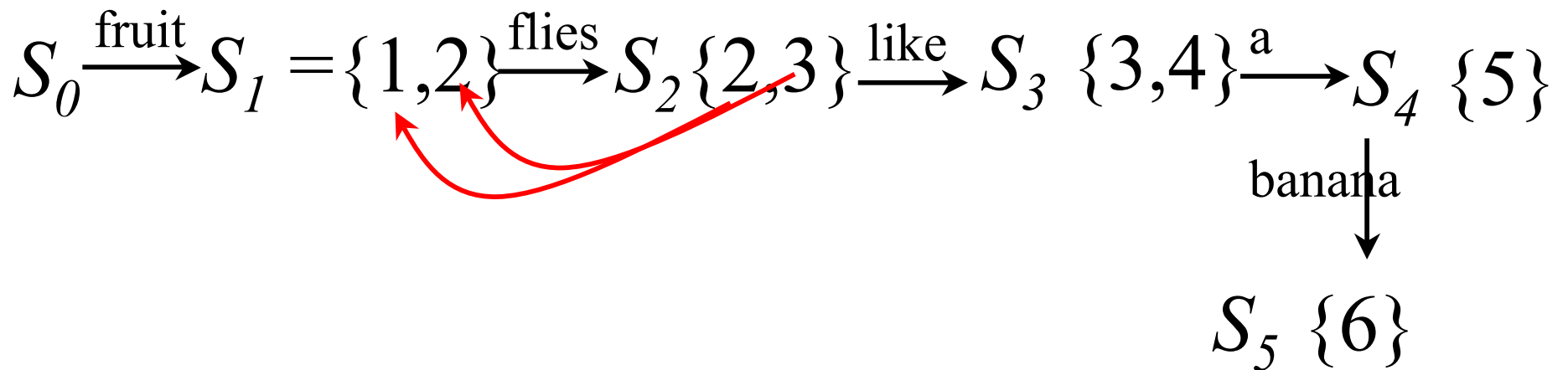
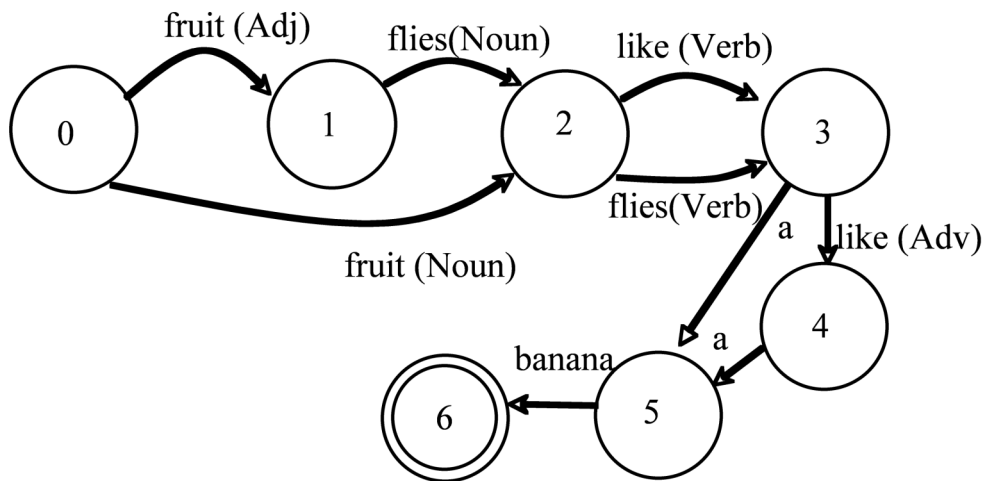
# FSA analog

‘fruit flies like a banana’



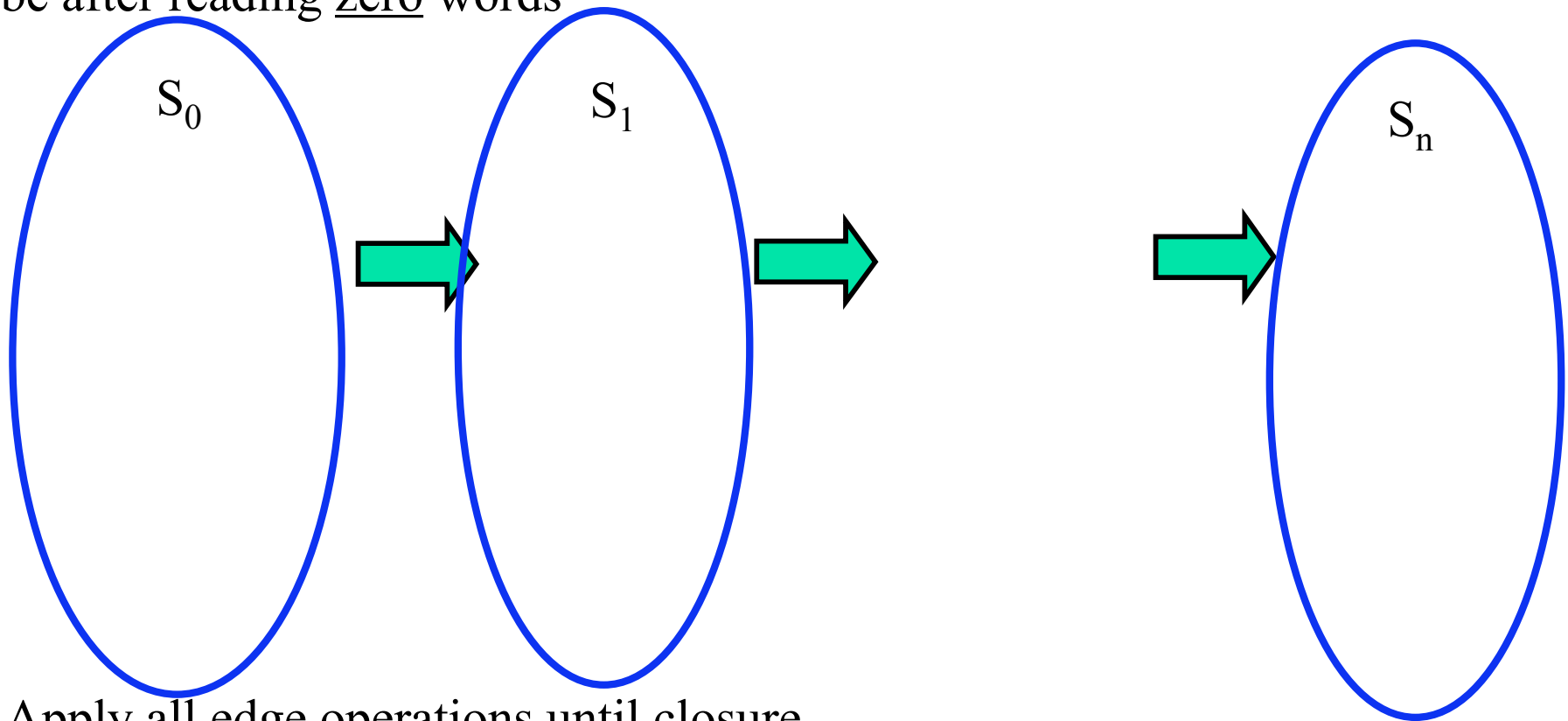
# As State Sets

## 'fruit flies like a banana'



# State set construction as sequence of machine states, extending paths...

Initial: state set  $S_0$  = where machine could be after reading zero words



Apply all edge operations until closure

# State Set machine

- Need to specify initial state
- Need to specify how to construct State Set  $S_i$ , given State Set  $S_{i-1}$
- Need to specify final state (accept or reject)
- For getting from one state to next, need only calculate how to scan under closure, word by word
- For the linear, or finite-state case, this is all easy
- For the hierarchical, or context-free case, we need a more complex representation of ‘state’, and how to get from one state set to the next

Table, or ‘chart’ as a sequence of State Sets  
 Column  $j$  = holds a *State Set*

What is a “state”? Now we need a bit more info...

0	$S \rightarrow \bullet NP VP$ $NP \rightarrow \bullet I$	$I \rightarrow \bullet$ $NP \rightarrow I \bullet$ $S \rightarrow NP \bullet VP$		$S \rightarrow NP VP \bullet$
1		$VP \rightarrow \bullet V NP$ $V \rightarrow \bullet shot$	$shot \rightarrow \bullet$ $V \rightarrow \bullet shot$ $VP \rightarrow V \bullet NP$	$VP \rightarrow V NP \bullet$
$i$ 2			$NP \rightarrow \bullet Det N$	$eleph \rightarrow \bullet$ $NP \rightarrow eleph \bullet$
3				
	0	1	2	3

# But not *much* more info...Earley's algorithm

- Use table as before; but 1 column *per* word as 'bag' (a 'chart')
- The column entries are called *states* or *items* and are represented with *dotted-rules*, plus information about *how much* of the rule has been found. A dotted rule is of 3 types:

$S \rightarrow \bullet VP$

A VP is predicted

$NP \rightarrow Det \bullet Noun$

An NP is in progress

$VP \rightarrow V NP \bullet$

A VP has been found

To the dotted rule we add 2 indices that denote where the phrase on the left of the arrow *starts* (its left edge), and *how far to the right* the phrase has been constructed *so far*

These two elements constitute a 'state'

So, e.g.,  $[NP \rightarrow Det \bullet Noun, (1, 2)]$  means that we started building an NP at word 1 of the input, and we've found the first word, a Det.

A collection of states in a given column (e.g., col. 2), is called a *State Set*

# Dotted rules: run-time conversion to binary branching

- $S \rightarrow \bullet VP$  [0,0]
- $NP \rightarrow Det \bullet Noun$  [1,2]
- $VP \rightarrow V NP \bullet$  [0,3]
- A VP is predicted at the start of the sentence
- An NP is in progress; the Det goes from 1 to 2
- A VP has been found starting at 0 and ending at 3

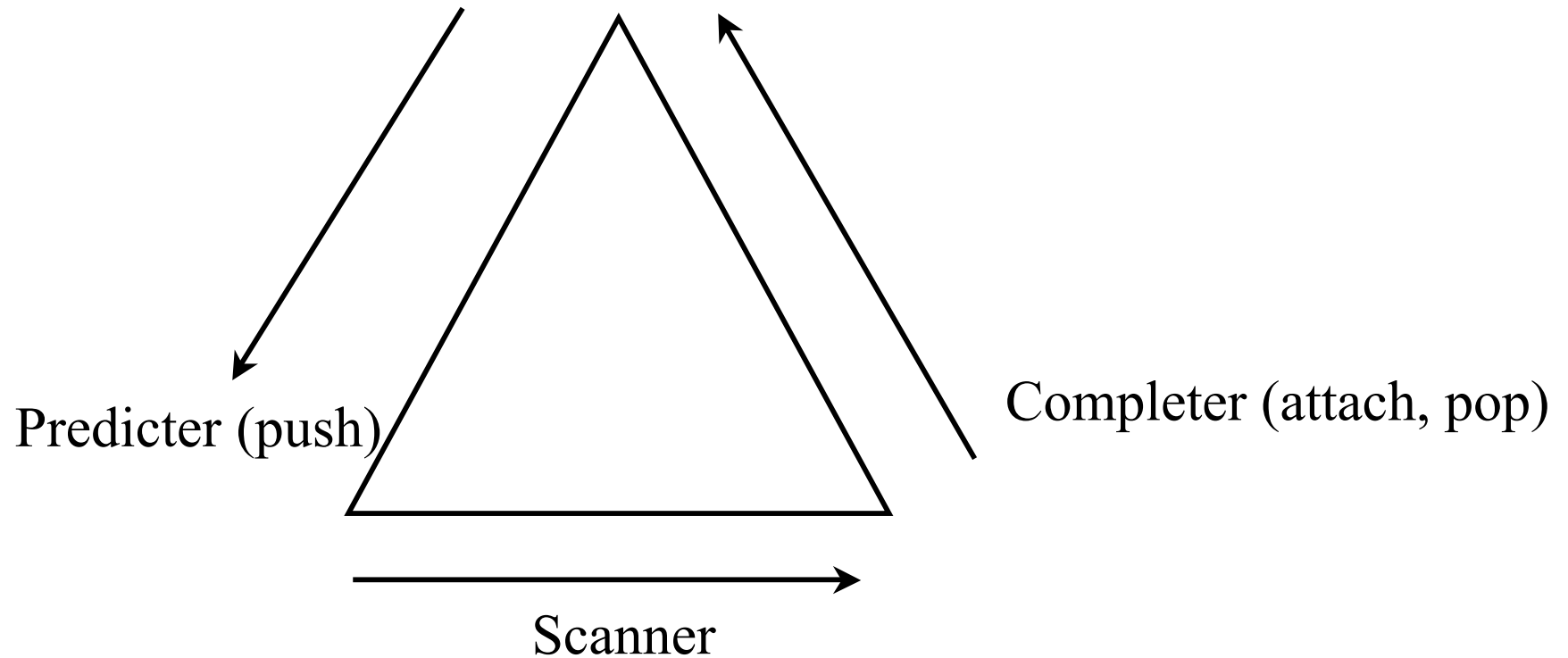
## What are...

- The initial state?  $S_0 = \{\text{Start} \rightarrow \bullet S, [0, 0]\}$
- The final state?  $S_n = \{\text{Start} \rightarrow S \bullet, [0, n]\}$
- How do we get from one State Set to the next?

We apply three operations on the items in a State Set until closure

Not just ‘scan’, but 2 more to *start* and *stop* the building of trees

Earley's algorithm uses 3 basic operations, corresponding to how we construct 'trees'



# Earley's algorithm (from textbook)

**function** EARLEY-PARSE(*words*, *grammar*) **returns** *chart*

ENQUEUE( $(\gamma \rightarrow \bullet S, [0, 0])$ , *chart*[0])

**for**  $i \leftarrow$  **from** 0 **to** LENGTH(*words*) **do**

**for each** *state* **in** *chart*[*i*] **do**

**if** INCOMPLETE?(*state*) **and**

      NEXT-CAT(*state*) is not a part of speech **then**

        PREDICTOR(*state*)

**elseif** INCOMPLETE?(*state*) **and**

      NEXT-CAT(*state*) is a part of speech **then**

        SCANNER(*state*)

**else**

      COMPLETER(*state*)

**end**

**end**

**return**(*chart*)

# The 3 operations: Predictor ('wishor'); Scanner, Completer

```
procedure PREDICTOR( $(A \rightarrow \alpha \bullet B \beta, [i, j])$ )  
  for each  $(B \rightarrow \gamma)$  in GRAMMAR-RULES-FOR( $B, grammar$ ) do  
    ENQUEUE( $(B \rightarrow \bullet \gamma, [j, j])$ ,  $chart[j]$ )  
  end  
  
procedure SCANNER( $(A \rightarrow \alpha \bullet B \beta, [i, j])$ )  
  if  $B \subset PARTS-OF-SPEECH(word[j])$  then  
    ENQUEUE( $(B \rightarrow word[j], [j, j+1])$ ,  $chart[j+1]$ )  
  
procedure COMPLETER( $(B \rightarrow \gamma \bullet, [j, k])$ )  
  for each  $(A \rightarrow \alpha \bullet B \beta, [i, j])$  in  $chart[j]$  do  
    ENQUEUE( $(A \rightarrow \alpha B \bullet \beta, [i, k])$ ,  $chart[k]$ )  
  end  
  
procedure ENQUEUE( $state, chart-entry$ )  
  if  $state$  is not already in  $chart-entry$  then  
    PUSH( $state, chart-entry$ )  
  end
```

## Algorithm is just a loop around this:

1. Add  $\text{Start} \rightarrow \bullet S$  to column 0
2. For each  $j$  from 0 to  $n$ :
  - i. For each dotted rule in column  $j$  (including those we add as we go), look at what's after the dot:
    - a. If word, *Scan*
    - b. If nonterminal, *Predict* (until closure)
    - c. If nothing after dot, *Complete* (until closure)
  - ii. Output 'yes' if  $\text{Start} \rightarrow S \bullet$  is in column  $n$

Note: Don't add duplicates! (AWP; but this is *tricky* to do in practice!)

Question: where is the *stack* in all of this? (Since we need a stack...)

## An Example...

ROOT → S

S → NP VP      NP → Papa

NP → Det N      N → caviar

NP → NP PP      N → spoon

VP → VP PP      V → ate

VP → V NP      P → with

PP → P NP      Det → the

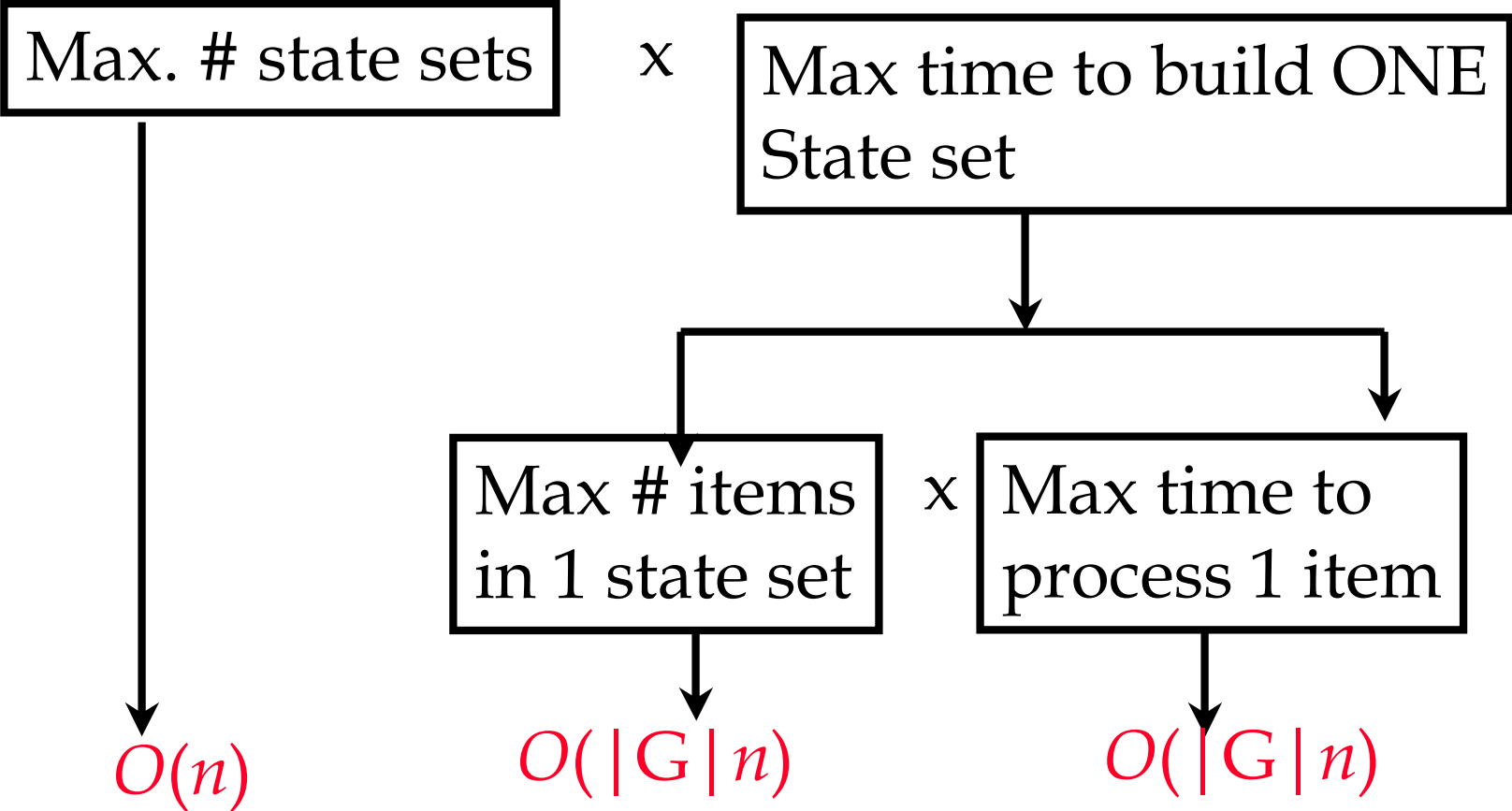
Det → a

Papa ate the caviar with a spoon

# Time complexity

- Algorithm iterates for every word of the input ( $n$  iterations)
- How many items can be created and processed in  $S_i$ ?
  - Each item in  $S_i$  has the form  $[A \rightarrow X_1 \dots \bullet C \dots X_m j]$ ,  $0 \leq j \leq i$
  - Thus  $O(n)$  items; better:
- The *scanner* and *predictor* operations on an item take at worst constant time (Why?)
- The *completer* operations on an item adds items of the form  $[B \rightarrow X_1 \dots \bullet A \dots X_m, l]$  to  $S_i$  with  $0 \leq l \leq i$ , so it may require up to  $O(n)$  time for each processed item
- Time required for each iteration  $S_i$  is thus  $O(n^2)$
- Time bound on entire algorithm is therefore  $O(n^3)$

So time complexity picture looks like this:

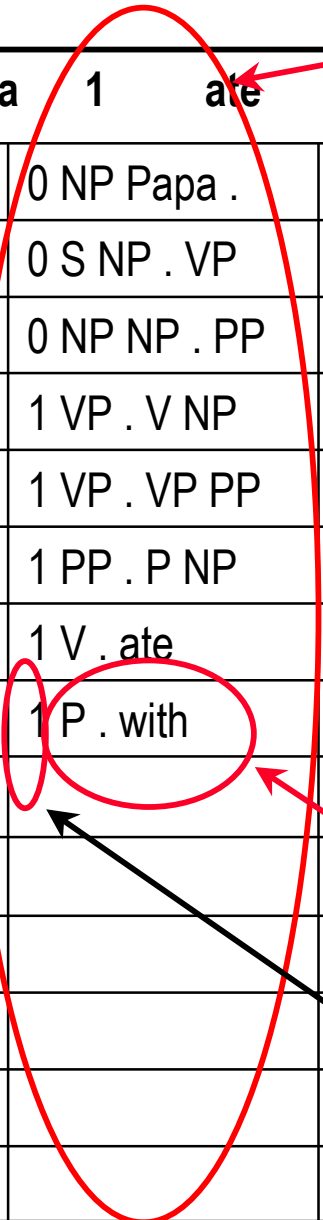




# Complexity

Max size of state set (column)  
 $O(|G| n)$

0	Papa	1	ate	2	the	3	caviar	4	with a spoon	7
0 ROOT . S	0 NP Papa .	1 V ate .	2 Det the .	3 N caviar .	...	6 N spoon .				
0 S . NP VP	0 S NP . VP	1 VP V . NP	2 NP Det . N	2 NP Det N .		5 NP Det N .				
0 NP . Det N	0 NP NP . PP	2 NP . Det N	3 N . caviar	1 VP V NP .		4 PP P NP .				
0 NP . NP PP	1 VP . V NP	2 NP . NP PP	3 N . spoon	2 NP NP . PP		5 NP NP . PP				
0 NP . Papa	1 VP . VP PP	2 NP . Papa		0 S NP VP .		2 NP NP PP .				
0 Det . the	1 PP . P NP	2 Det . the		1 VP VP . PP		1 VP VP PP .				
0 Det . a	1 V . ate	2 Det . a		4 PP . P NP		7 PP . P NP				
	1 P . with			0 ROOT S .		1 VP V NP .				
				4 P . with		2 NP NP . PP				
						0 S NP VP .				
						1 VP VP . PP				
						7 P . with				
						<b>0 ROOT S .</b>				



|G|

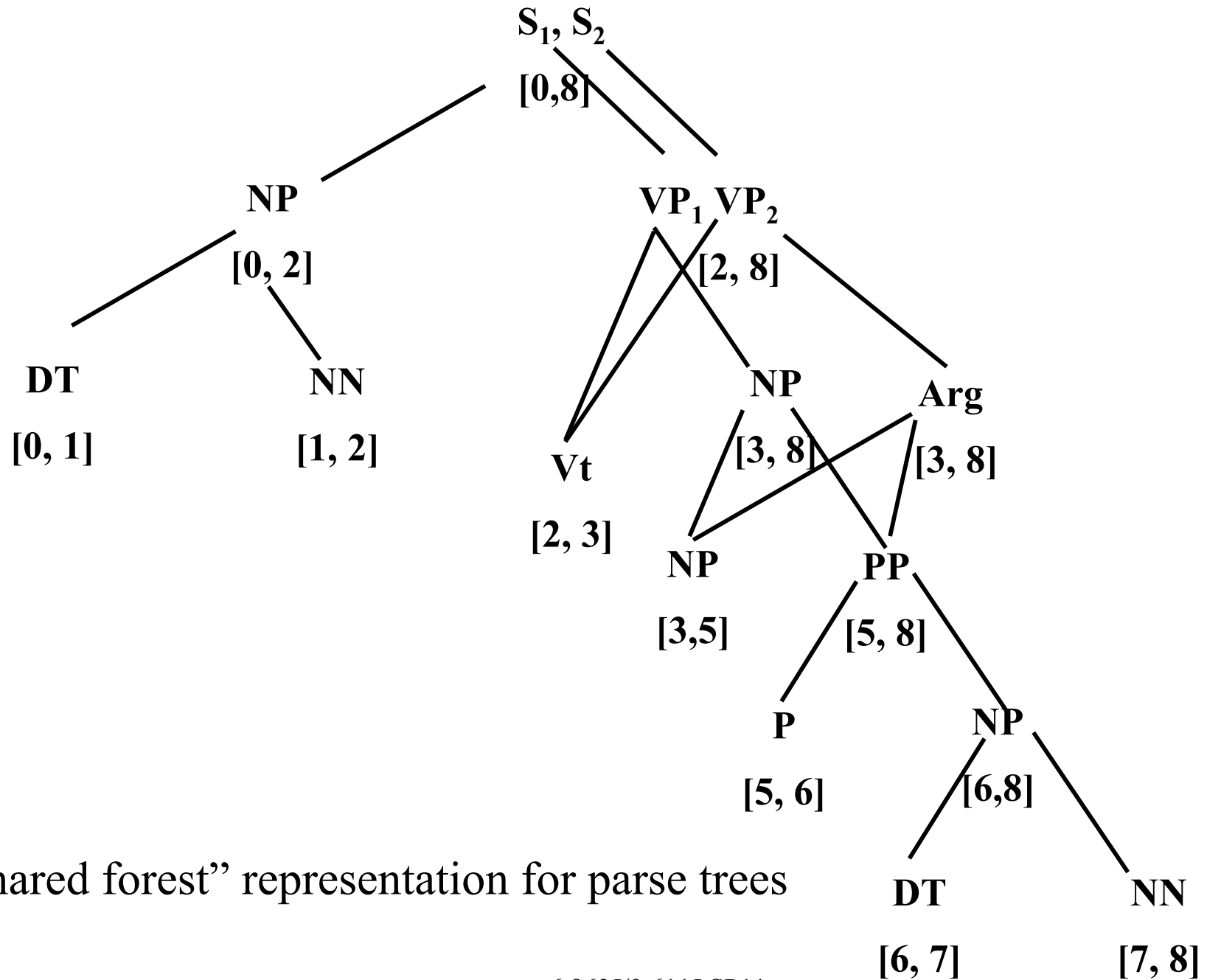
n

**0 ROOT S .**

# Max time to process any *one* item in a state set

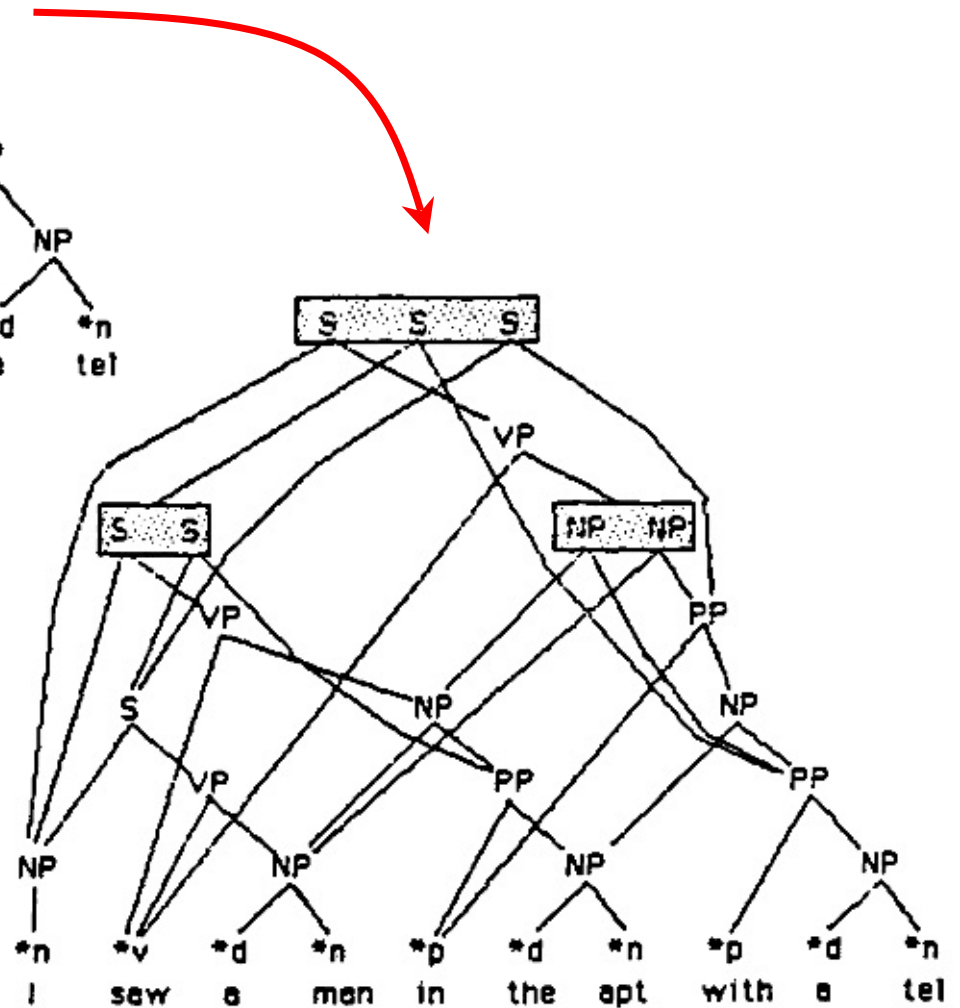
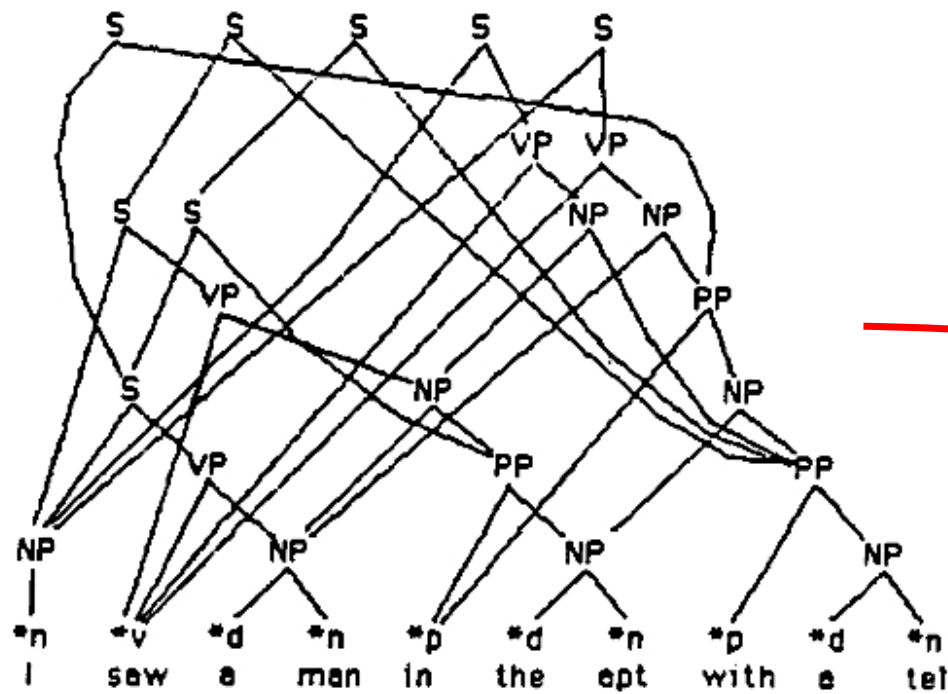
0	Papa	1	ate	2	the	3	caviar	4	with a spoon	7
0 ROOT . S	0 NP Papa .	1 V ate .	2 Det the .	3 N caviar .	...	6 N spoon .				
0 S . NP VP	0 S NP . VP	1 VP V . NP	2 NP Det . N	2 NP Det N .		5 NP Det N .				
0 NP . Det N	0 NP NP . PP	2 NP . Det N	3 N . caviar	1 VP V NP .		4 PP P NP .				
0 NP . NP PP	1 VP . V NP	2 NP . NP PP	3 N . spoon	2 NP NP . PP		5 NP NP . PP				
0 NP . Papa	1 VP . VP PP	2 NP Papa		0 S NP VP .		2 NP NP PP .				
0 Det . the	1 PP . P NP	2 Det . the		1 VP VP . PP		1 VP VP PP .				
0 Det . a	1 V . ate	2 Det . a		4 PP . P NP		7 PP . P NP				
	1 P . with			0 ROOT S .		1 VP V NP .				
				4 P . with		2 NP NP . PP				
						0 S NP VP .				
						1 VP VP . PP				
						7 P . with				
						0 ROOT S .				

Go back to column  $i$  &  
Advance dot in all  
items in column,  
so  $O(|G|n)$



“shared forest” representation for parse trees

We can pack parses more tightly...



# Implementation Details: we could have a whole algorithms course on this...

- Adding entry to parse take must check in  $O(1)$  time whether another copy is there
- You have only  $O(1)$  time to add entry if is new, so must be able to find bottom of the right column quickly
- (For probabilistic version): must keep track of that entry's current best parse, and total weight
- Rule  $A \rightarrow WXYZ$  represented as list  $(W, X, Z, A)$
- The dotted rule  $A \rightarrow WX \bullet YZ$  represented as pair  $(2, R)$  where  $R$  is a pointer to the rule (alternatively: list only elts that have not yet been matched, throw away  $W$  and  $X$  after matching; this keeps parse table smaller)
- Each column in parse table as extensible list or linked list with a tail pointer
- Duplicate check: use hash table

# Speed techniques

- Keep track of which categories have already been PREDICTed for the current column
- Build a trie that allows you to represent everything of the form  $(3, A \rightarrow B \bullet C \dots)$  as a single entry in the parse table (there's a better idea, in a bit...)

# Left-Corner Parsing (“Left ancestor parsing”)

- Technique for 1 word of lookahead in algorithms like Earley’s
- (can also do multi-word lookahead but it’s harder)

# Basic Earley's Algorithm

0	Papa	1
0 ROOT . S		0 NP Papa .
0 S . NP VP		0 S NP . VP
0 NP . Det N		0 NP NP . PP
0 NP . NP PP		
0 NP . Papa		
0 Det . the		
0 Det . a		

**attach**

0	Papa	1
0 ROOT . S		0 NP Papa .
0 S . NP VP		0 S NP . VP
0 NP . Det N		0 NP NP . PP
0 NP . NP PP		1 VP . V NP
0 NP . Papa		1 VP . VP PP
0 Det . the		
0 Det . a		

**predict**

0	Papa	1
0 ROOT . S		0 NP Papa .
0 S . NP VP		0 S NP . VP
0 NP . Det N		0 NP NP . PP
0 NP . NP PP		1 VP . V NP
0 NP . Papa		1 VP . VP PP
0 Det . the		1 PP . P NP
0 Det . a		

**predict**

0	Papa	1
0 ROOT . S		0 NP Papa .
0 S . NP VP		0 S NP . VP
0 NP . Det N		0 NP NP . PP
0 NP . NP PP		1 VP . V NP
0 NP . Papa		1 VP . VP PP
0 Det . the		1 PP . P NP
0 Det . a		1 V . ate
		1 V . drank
		1 V . snorted

## predict

- V makes us add all the verbs in the vocabulary!
- **Slow** – we'd like a shortcut.

0	Papa	1
0 ROOT . S		0 NP Papa .
0 S . NP VP		0 S NP . VP
0 NP . Det N		0 NP NP . PP
0 NP . NP PP		1 VP . V NP
0 NP . Papa		1 VP . VP PP
0 Det . the		1 PP . P NP
0 Det . a		1 V . ate
		1 V . drank
		1 V . snorted

## predict

- Every •VP adds all VP  $\rightarrow$  ... rules again.
- Before adding a rule, check it's not a duplicate.
- Slow if there are  $> 700$  VP  $\rightarrow$  ... rules, so what will you do? (A BIG constant ow!)
- Keep track of which categories have already been Predicted for a current column
- Add a table indexed on nonT's w/ 1 bit to flag 'added'; clear this going to next column
- Or, a prefix table  $R(A,B)$  the set of all  $A$  s.t. there is at least one grammar rule  $A \rightarrow B$ .

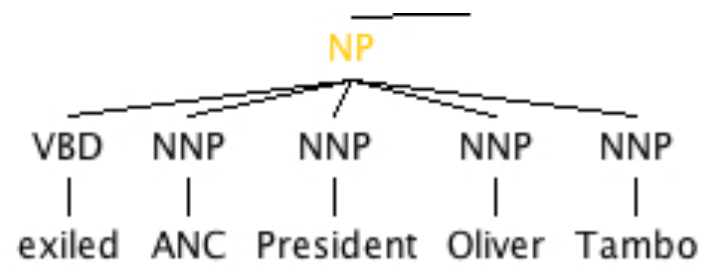
There can be a lot of such rules

- Consider extracting rules from the 39,000 sentence Wall Street Journal corpus
- There are 694 rules that expand NPs, from  
NP → PUNC: NN NN NN PUNC.

to...

NP → VBN NNS NNS NNS NNS

This is a BIG constant...



0	Papa	1
0 ROOT . S		0 NP Papa .
0 S . NP VP		0 S NP . VP
0 NP . Det N		0 NP NP . PP
0 NP . NP PP		1 VP . V NP
0 NP . Papa		1 VP . VP PP
0 Det . the		1 PP . P NP
0 Det . a		1 V . ate
		1 V . drank
		1 V . snorted
		1 P . with

## predict

- P makes us add all the prepositions ...

# 1-word lookahead would help

0	Papa	1	ate
0 ROOT . S		0 NP Papa .	
0 S . NP VP		0 S NP . VP	
0 NP . Det N		0 NP NP . PP	
0 NP . NP PP		1 VP . V NP	
0 NP . Papa		1 VP . VP PP	
0 Det . the		1 PP . P NP	
0 Det . a		1 V . ate	
		1 V . drank	
		<del>1 V . snorted</del>	
		<del>1 P . with</del>	

No point in adding words other than ate

# 1-word lookahead would help

0	Papa	1	ate
0 ROOT . S		0 NP Papa .	
0 S . NP VP		0 S NP . VP	
0 NP . Det N		<del>0 NP NP . PP</del>	
0 NP . NP PP		1 VP . V NP	
0 NP . Papa		1 VP . VP PP	
0 Det . the		<del>1 PP . P NP</del>	
0 Det . a		1 V . ate	
		<del>1 V . drank</del>	
		<del>1 V . snorted</del>	
		<del>1 P . with</del>	

In fact, no point in adding any constituent that can't start with ate  
 Don't bother adding PP, P, etc.

No point in adding words other than ate

# With Left-Corner Filter

0	Papa	1	ate
0 ROOT . S		0 NP Papa .	
0 S . NP VP		0 S NP . VP	
0 NP . Det N		<del>0 NP NP . PP</del>	
0 NP . NP PP			
0 NP . Papa			
0 Det . the			
0 Det . a			

**attach**

PP can't start with ate

Birth control – now we won't predict

1 PP . P NP

1 P . with

either!

Need to know that ate can't start PP

Take closure of all categories that it does start ...

0	Papa	1	ate
0 ROOT . S		0 NP Papa .	
0 S . NP VP		0 S NP . VP	
0 NP . Det N		<del>0 NP NP . PP</del>	
0 NP . NP PP		1 VP . V NP	
0 NP . Papa		1 VP . VP PP	
0 Det . the			
0 Det . a			

**predict**

0	Papa	1	ate
0 ROOT . S		0 NP Papa .	
0 S . NP VP		0 S NP . VP	
0 NP . Det N		<del>0 NP NP . PP</del>	
0 NP . NP PP		1 VP . V NP	
0 NP . Papa		1 VP . VP PP	
0 Det . the		1 V . ate	
0 Det . a		<del>1 V . drank</del>	
		<del>1 V . snorted</del>	

**predict**

0	Papa	1	ate
0 ROOT . S	0 NP Papa .		
0 S . NP VP	0 S NP . VP		
0 NP . Det N	<del>0 NP NP . PP</del>		
0 NP . NP PP	1 VP . V NP		
0 NP . Papa	1 VP . VP PP		
0 Det . the	1 V . ate		
0 Det . a	<del>1 V . drank</del>		
	<del>1 V . snorted</del>		

**predict**

## Implementing via 3 hash tables, $R$ , $P$ , $S$

- Done when reading in the grammar:  $R$ ,  $P$
- The prefix table  $R(A,B)$  stores the set of all grammar rules of the form  $A \rightarrow B \dots$
- The left parent table  $P: P(B)$  stores all  $A$  s.t. there is at least one grammar rule of the form  $A \rightarrow B \dots$  ( $B$  is the ‘left child’ of  $A$ )
- When you read a grammar rule of the form  $A \rightarrow B \dots$  you simply add  $A$  to  $P(B)$  iff  $R(A,B) = \emptyset$  (this test avoids duplicates in  $P(A,B)$ ), then add the rule itself to  $R(A,B)$
- Let  $w_j$  be the word that starts at position (column)  $j$
- Before you begin to process entries in column  $j$ , construct a third hash table  $S_j$  only used in processing that column, the left ancestor pair table,  $S_j(A)$

# Left-corner parsing

- Left ancestor pair table  $S_j(A)$  stores the set of all  $B$  s.t.  $A$  is a left parent of  $B$  and  $B$  is a left-ancestor of  $w_j$
- That is,  $A \in P(B)$ , and either  $B = w_j$  or  $B \in P(w_j)$  or  $B \in P(P(w_j))$  or ...
- It is straightforward to compute  $S_j$  by depth-first search: process some  $Y$ , initially  $w_j$ , by adding  $Y$  to  $S_j(X)$ , for each  $X \in P(Y)$ .
- Where this is the first addition to  $S_j(X)$ , recursively process  $X$  (why only the first?)
- An example...

Example:  $w_j = \textit{lead}$  (either N or V)

- $P(w_j) = \{N, V\}$ , suppose grammar is s.t.  
 $P(N) = \{\text{NP}\}$ ,  $P(V) = \{\text{VP}\}$   
 $P(\text{NP}) = \{\text{NP}, \text{S}\}$  (so NP is first child of either NP or S)  
 $P(\text{VP}) = \{\text{VP}\}$  (so VP can be the first child only of VP)  
 $P(\text{S}) = \{\}$  (so S can't be first child of anything).

# Given this table $P$ and $R$

Then:

$S_j(N) = \{lead\}$ , so  $Predict(N)$  adds all  $N \rightarrow \bullet lead$  rules via  $R(N, lead)$

$S_j(V) = \{lead\}$ , so  $Predict(V)$  adds all  $V \rightarrow \bullet lead$  rules via  $R(N, lead)$

$S_j(NP) = \{N, NP\}$  so  $Predict(NP)$  adds all  $NP \rightarrow \bullet N$  rules via  $R(N, NP)$   
and all  $NP \rightarrow \bullet NP$  rules via  $R(NP, NP)$

but does not add any  $NP \rightarrow \bullet Det$  NP rules, since *lead* can't be the first word of a Det

$S_j(VP) = \{V, VP\}$  so  $Predict(VP)$  adds all  $VP \rightarrow \bullet V$  rules via  $R(VP, V)$

adds all  $VP \rightarrow \bullet VP$  rules via  $R(VP, VP)$

$S_j(S) = \{NP\}$  so  $Predict(S)$  adds all  $S \rightarrow \bullet NP$  rules via  $R(S, NP)$

but does not add any  $S \rightarrow \bullet PP$  rules since *lead* can't be the first word of a PP

# Why does this matter? Grammar gets big...

	CT Grammar	ATIS Grammar	PT Grammar
Rules	24,456	4,592	15,039
Nonterminals	3,946	192	38
Terminals	1,032	357	47
# Test Sentences	162	98	30
Average Length	8.3	11.4	5.7
# Grammatical	150	70	30
Average # Parses	5.4	940	$7.2 \times 10^{27}$

# Comparison of methods

	CT Grammar	ATIS Grammar	PT Grammar
LC <sub>2</sub> +BUPM	3.1	7.0	27.0
CKY	25.0	7.7	50.9
E/GHR	7.3	8.6	27.7
GLR(0)	3.2	14.0	timed out
LC+follow	2.4	6.6	29.6
GLR(0)+follow	2.3	14.1	timed out
GLALR(1)	3.8	14.7	—

Annotations:

- Earley (points to the first column)
- 1 token lookahead (points to the 'GLR(0)+follow' row)
- 't-d' left-corner (points to the 'PT Grammar' column)
- strict b-u (points to the 'PT Grammar' column)

## Some filtering like this is necessary

- Example: consider a ‘wsj’ grammar, made from just ‘reading off’ the rules in the 40,000 example sentences of the Wall Street Journal corpus

- There are 694 NP expansion rules!

NP → PUNC: NN NN NN PUNC.

...

NP → DT

NP → DT ADJP , JJ NN

...

NP → VBN NNP NNP NNP NNP (where’s this come from?)

# Merging Right-Hand Sides

- Grammar might have rules
$$\mathbf{X \rightarrow A G H P}$$
$$\mathbf{X \rightarrow B G H P}$$
- Could end up with both of these in chart:
$$(2, \mathbf{X \rightarrow A . G H P})$$
 in column 5
$$(2, \mathbf{X \rightarrow B . G H P})$$
 in column 5
- But these are now interchangeable: if one produces  $X$  then so will the other
- To avoid this redundancy, can always use dotted rules of this form:  $\mathbf{X \rightarrow \dots G H P}$

## Merging Right-Hand Sides

- Similarly, grammar might have rules  
 $X \rightarrow A G H P$   
 $X \rightarrow A G H Q$
- Could end up with both of these in chart:  
 $(2, X \rightarrow A . G H P)$  in column 5  
 $(2, X \rightarrow A . G H Q)$  in column 5
- Not interchangeable, but we'll be processing them in parallel for a while ...
- Solution: write grammar as  $X \rightarrow A G H (P|Q)$

# Merging Right-Hand Sides

- Combining the two previous cases:

$X \rightarrow A G H P$

$X \rightarrow A G H Q$

$X \rightarrow B G H P$

$X \rightarrow B G H Q$

becomes

$X \rightarrow (A \mid B) G H (P \mid Q)$

- And often nice to write stuff like

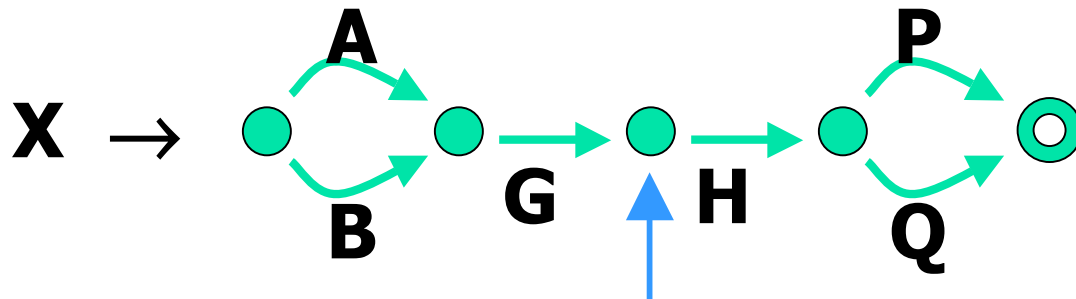
$NP \rightarrow (\text{Det} \mid \varepsilon) \text{Adj}^* N$

# Merging Right-Hand Sides

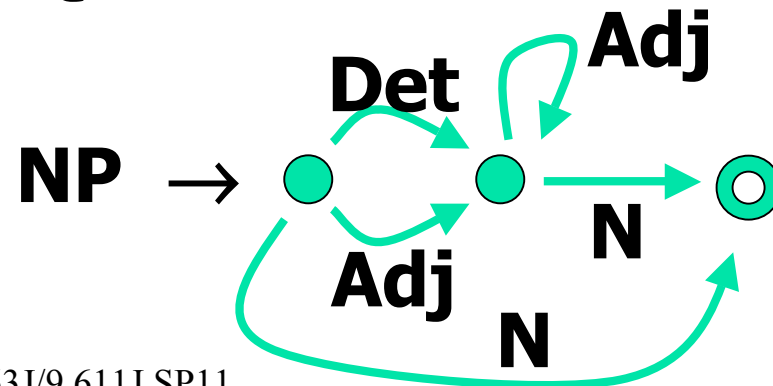
$X \rightarrow (A \mid B) G H (P \mid Q)$

$NP \rightarrow (\text{Det} \mid \epsilon) \text{Adj}^* N$

- These are regular expressions!
- Build their minimal DFAs:



- Automaton states replace dotted rules ( $X \rightarrow A G . H P$ )



# Merging Right-Hand Sides

Indeed, *all* NP → rules can be unioned into a single DFA!

NP → ADJP ADJP JJ JJ NN NNS

NP → ADJP DT NN

NP → ADJP JJ NN

NP → ADJP JJ NN NNS

NP → ADJP JJ NNS

NP → ADJP NN

NP → ADJP NN NN

NP → ADJP NN NNS

NP → ADJP NNS

NP → ADJP NPR

NP → ADJP NPRS

NP → DT

NP → DT ADJP

NP → DT ADJP , JJ NN

NP → DT ADJP ADJP NN

NP → DT ADJP JJ JJ NN

NP → DT ADJP JJ NN

NP → DT ADJP JJ NN NN

**etc.**