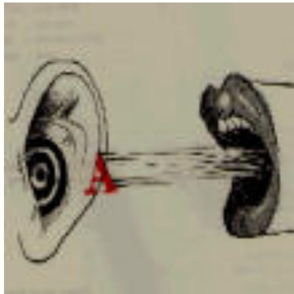
Lecture 8: Parsing tricks; beyond context-free grammars



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

The Menu Bar

- Administrivia: Lab 3…
- The Earley algorithm
- Some speedup tricks
- How do the actual ‘large scale systems’ do?
- Pluses and minuses of context-free grammars

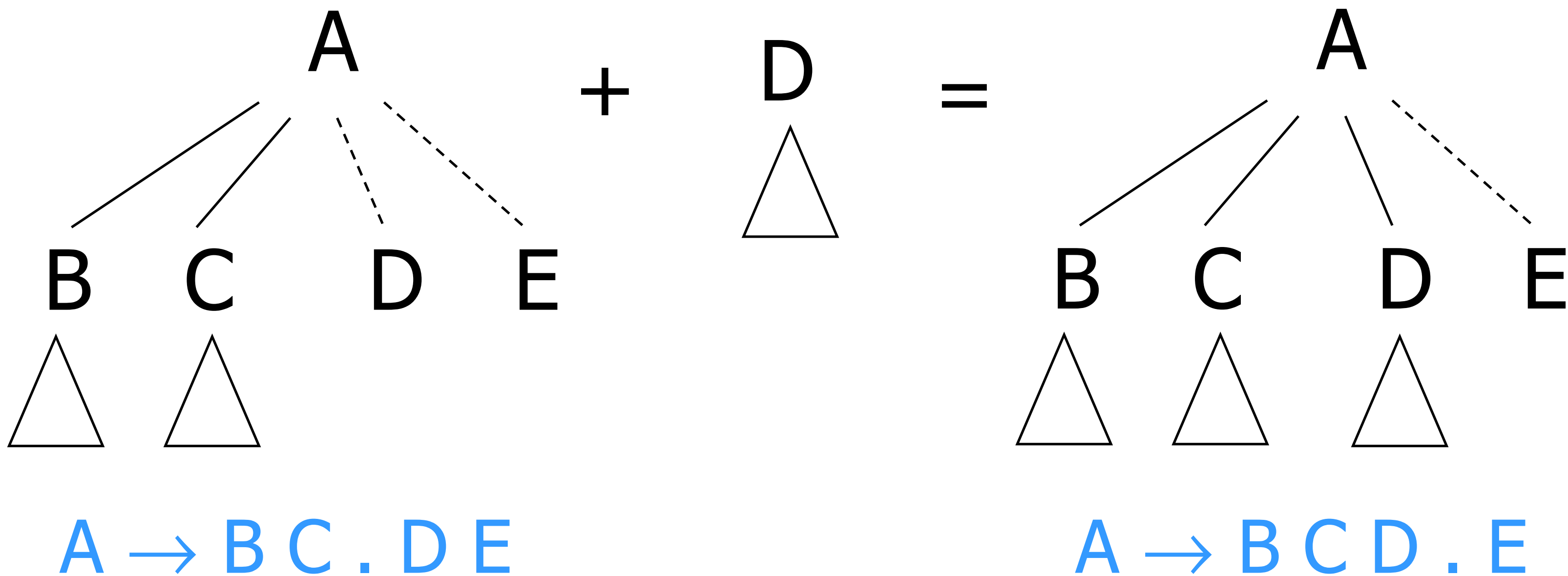
Earley parsing

- <u>Top down</u> 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., NP→ 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 <u>partial</u> constituents in input
  - A → B C . D E is partial: only the first half of the A



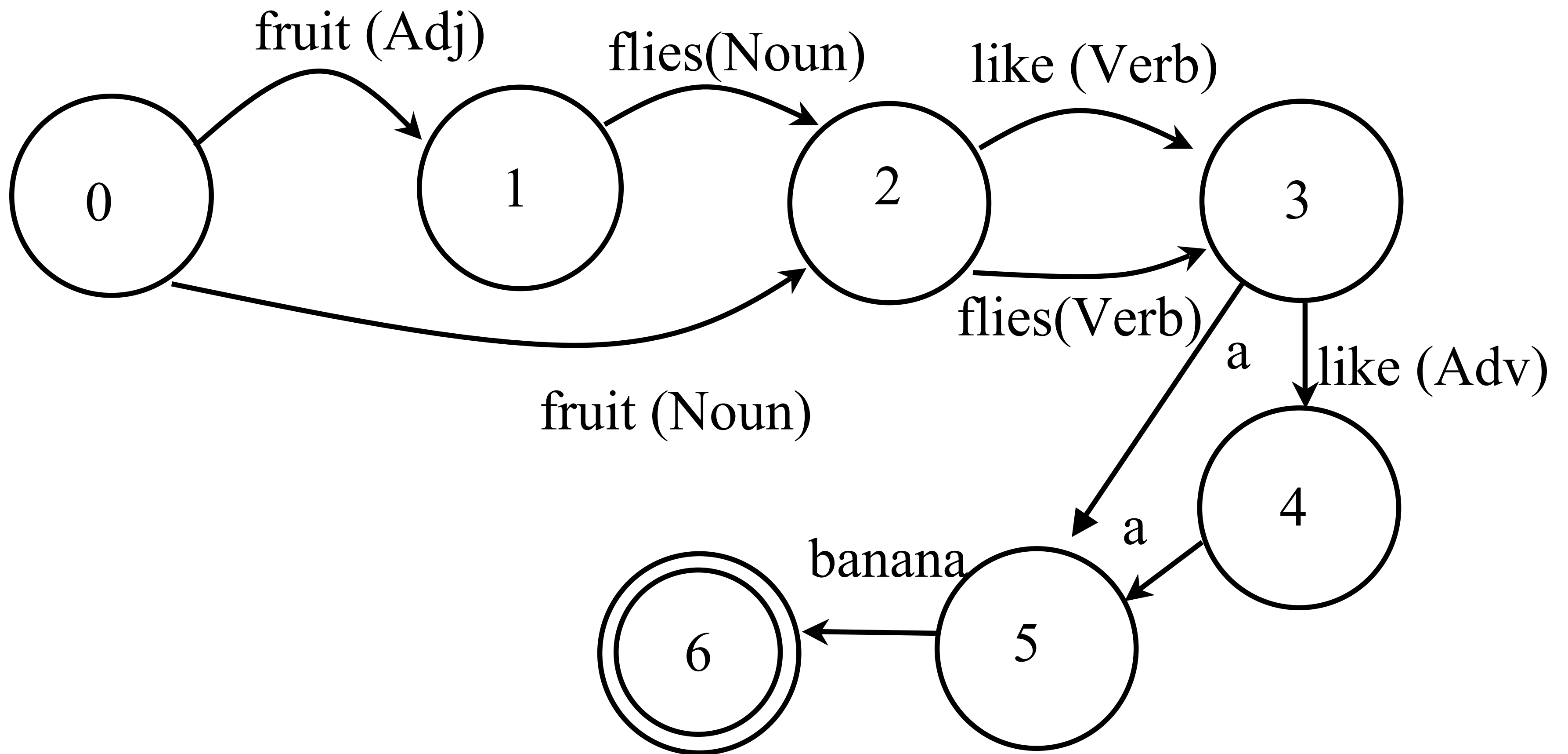
Overview of Earley's Algorithm

- 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 → B C . D E gives A → B C D . E
  - Attaching E to that gives A → B C D E .
  - Now we have a complete A that we can attach to hypotheses immediately preceding the A, etc.

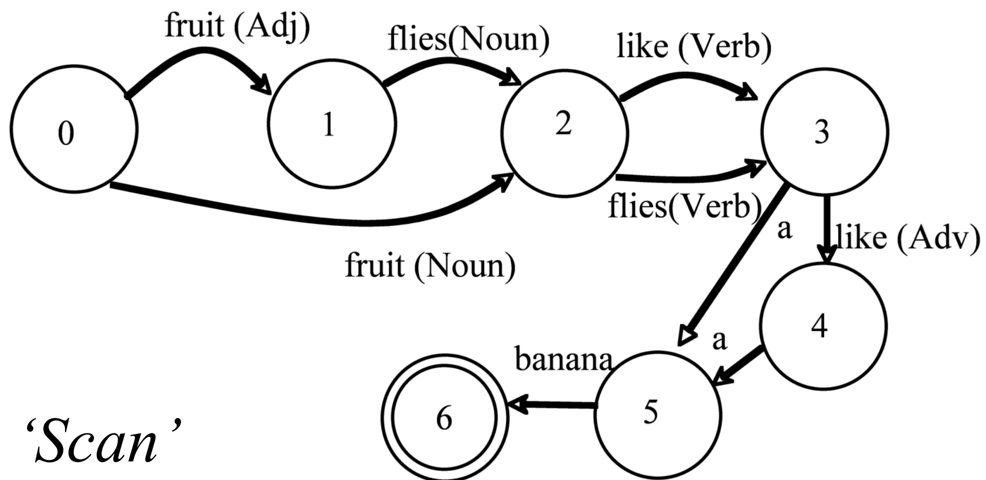
Motivation

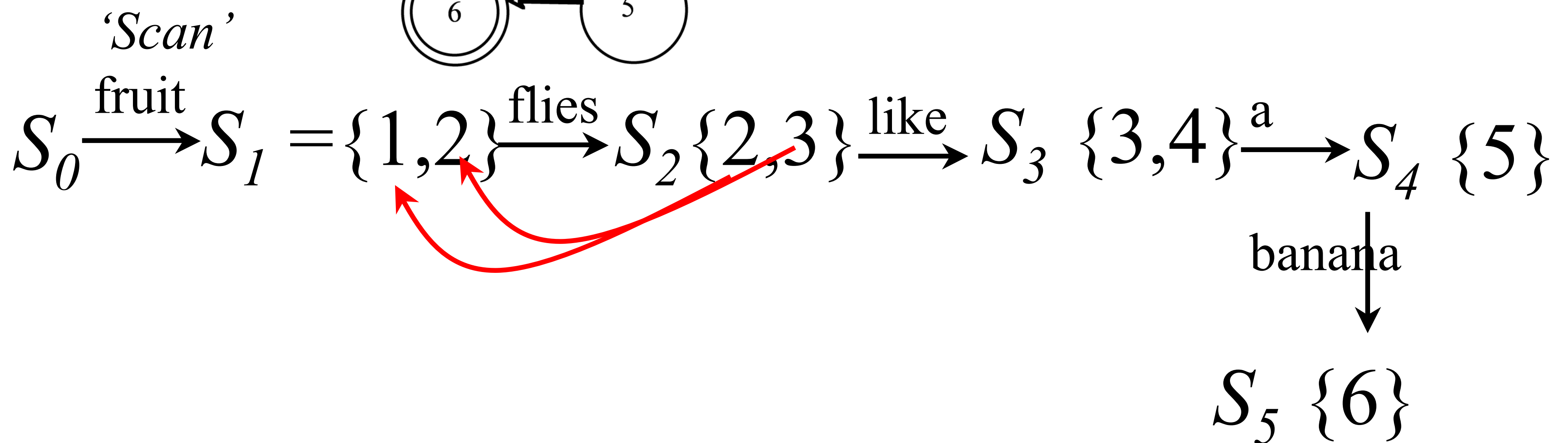
- We will carry out a *deterministic* simulation of a *nondeterministic* machine that can keep track of 'all the possible next states' S_i it can be in after reading each word
- So we imagine a sequence of state sets, S_0, S_1…
- Representation as a 'chart' of 'columns'; Column *i* or *chart*(*i*) will contain *all of the possible* machine states after reading the ith 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'



Parsing with State Sets
‘fruit flies like a banana’

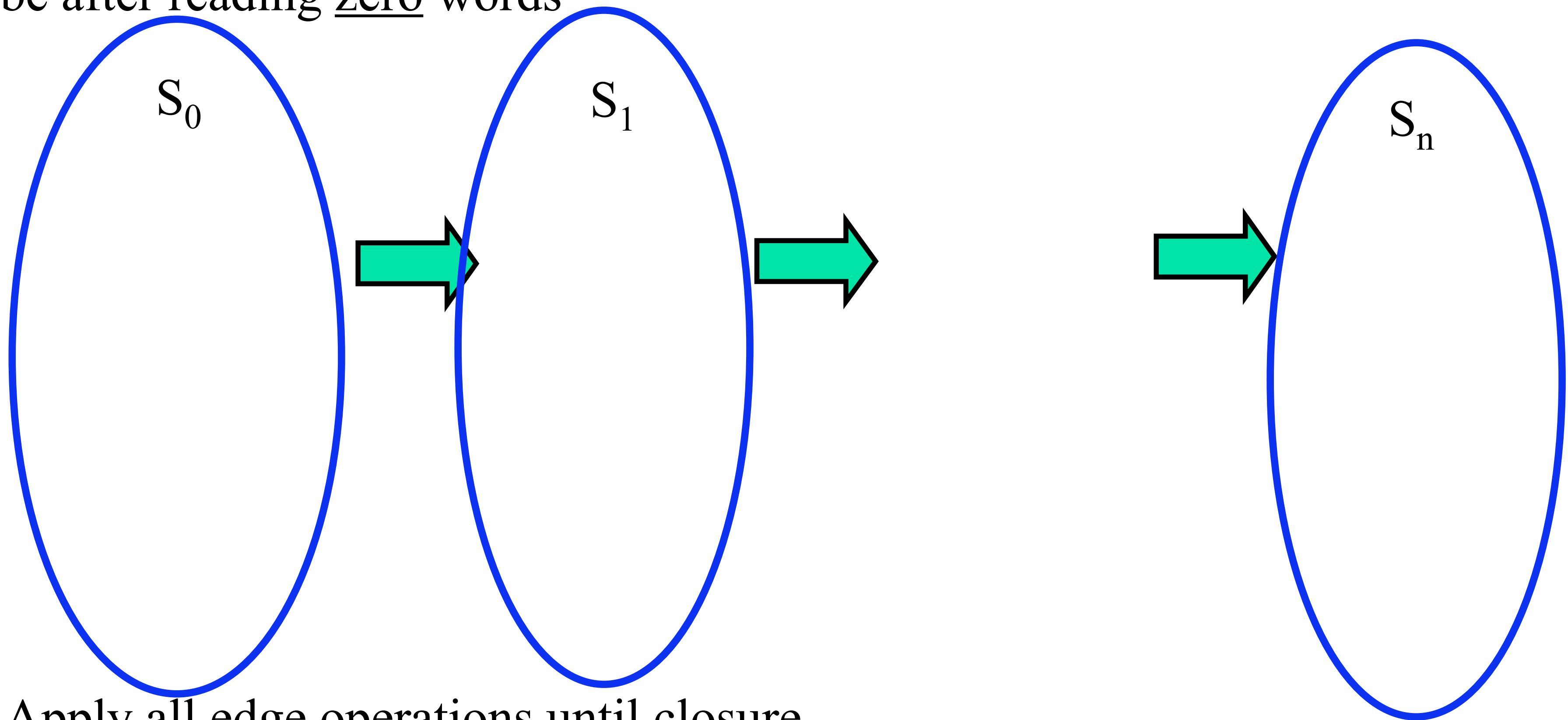




State set construction as sequence of machine states, extending paths…

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…

i \ j	0	1	2	3
0	S → • NP VP NP → • I	I → • NP →I • S → NP • VP		S → NP VP •
1		VP → • V NP V → • shot	shot → • V → • shot VP → V • NP	VP → V NP •
2			NP → •Det N	eleph → • NP → eleph•
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 → ● VP [0,0]
- NP → Det ● Noun [1,2]
- VP → V NP ● [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

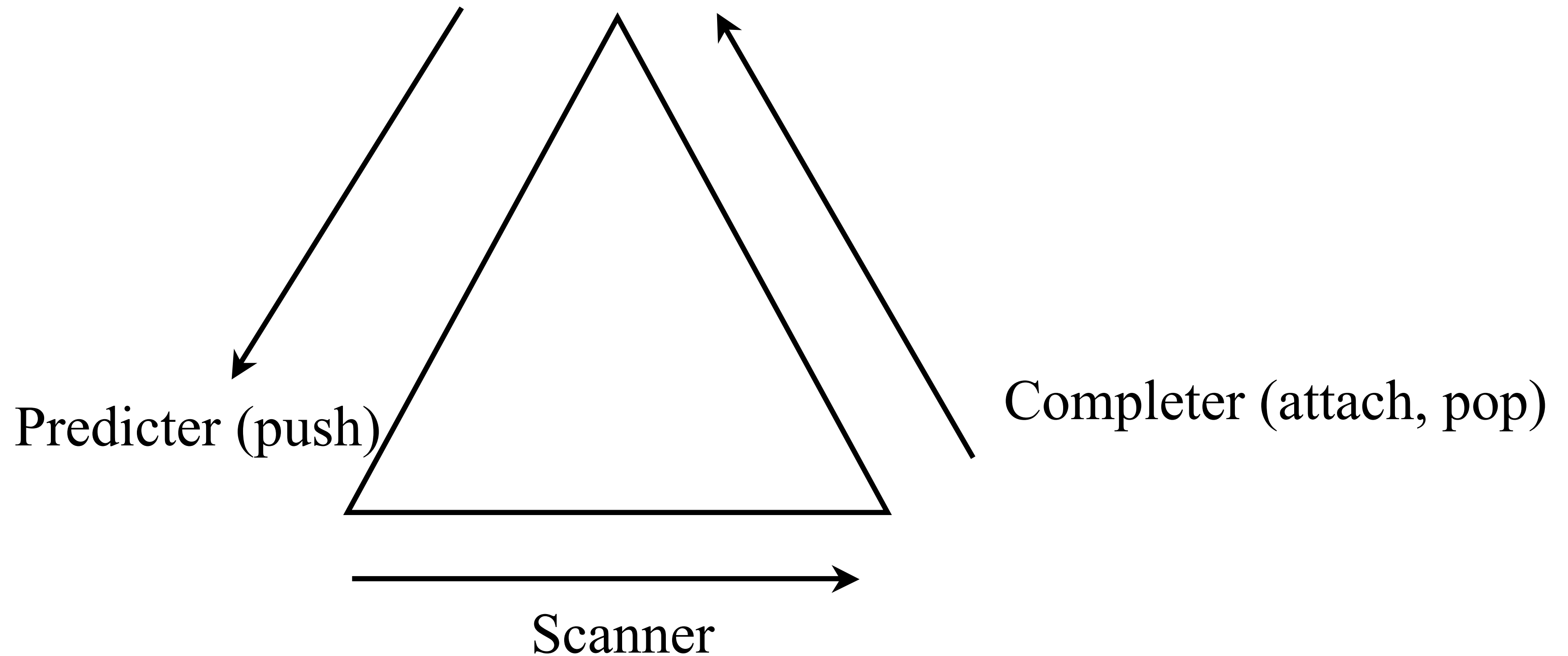
What are…

- The initial state? S_0={Start→•S, [0,0]}
- The final state? S_n={Start →S •, [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((γ → • S, [0,0]), chart[0])
  for i ← 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 → α • B β, [i, j]))
    for each (B → γ) in GRAMMAR-RULES-FOR(B, grammar) do
        ENQUEUE((B → • γ, [j, j]), chart[j])
    end

procedure SCANNER((A → α • B β, [i, j]))
    if B ⊂ PARTS-OF-SPEECH(word[j]) then
        ENQUEUE((B → word[j], [j, j+1]), chart[j+1])

procedure COMPLETER((B → γ •, [j,k]))
    for each (A → α • B β, [i, j]) in chart[j] do
        ENQUEUE((A → α B • β, [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 Start → • 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 Start → S • 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…

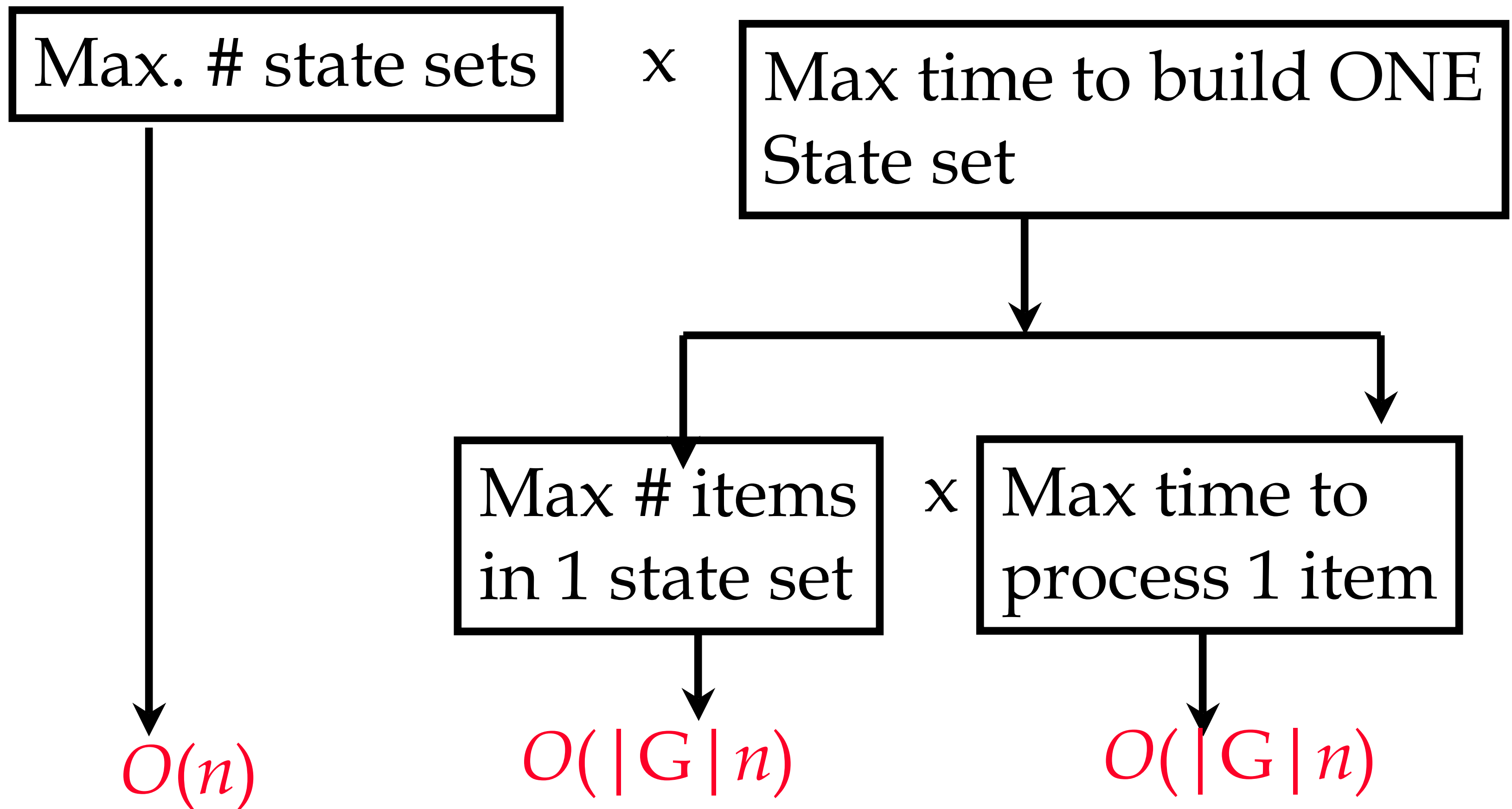
ROOT	$\rightarrow$	S			
S	$\rightarrow$	NP VP	NP	$\rightarrow$	Papa
NP	$\rightarrow$	Det N	N	$\rightarrow$	caviar
NP	$\rightarrow$	NP PP	N	$\rightarrow$	spoon
VP	$\rightarrow$	VP PP	V	$\rightarrow$	ate
VP	$\rightarrow$	V NP	P	$\rightarrow$	with
PP	$\rightarrow$	P NP	Det	$\rightarrow$	the
			Det	$\rightarrow$	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 $[A \rightarrow X_1 \ldots \bullet C \ldots X_m, j]$, $0 \leq j \leq i$
  - 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 $[B \rightarrow X_1 \ldots \bullet A \ldots X_m, l]$ 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

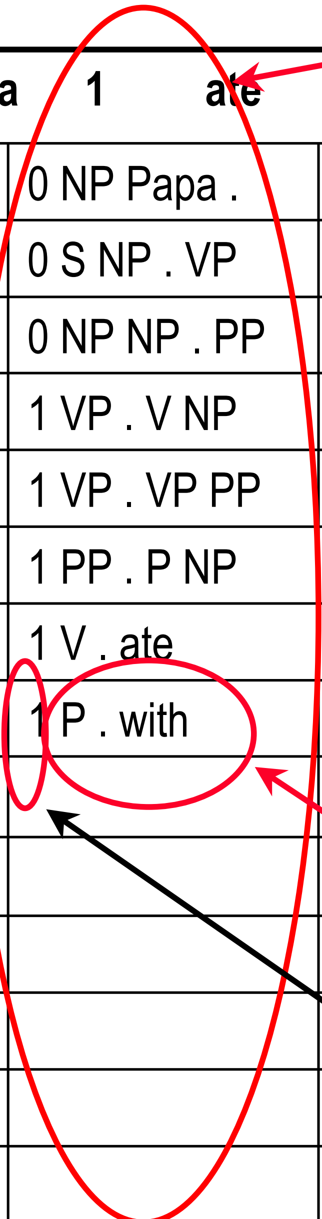
Max # of state sets = $O(n)$

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
						0 ROOT S .

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
					0 ROOT S .



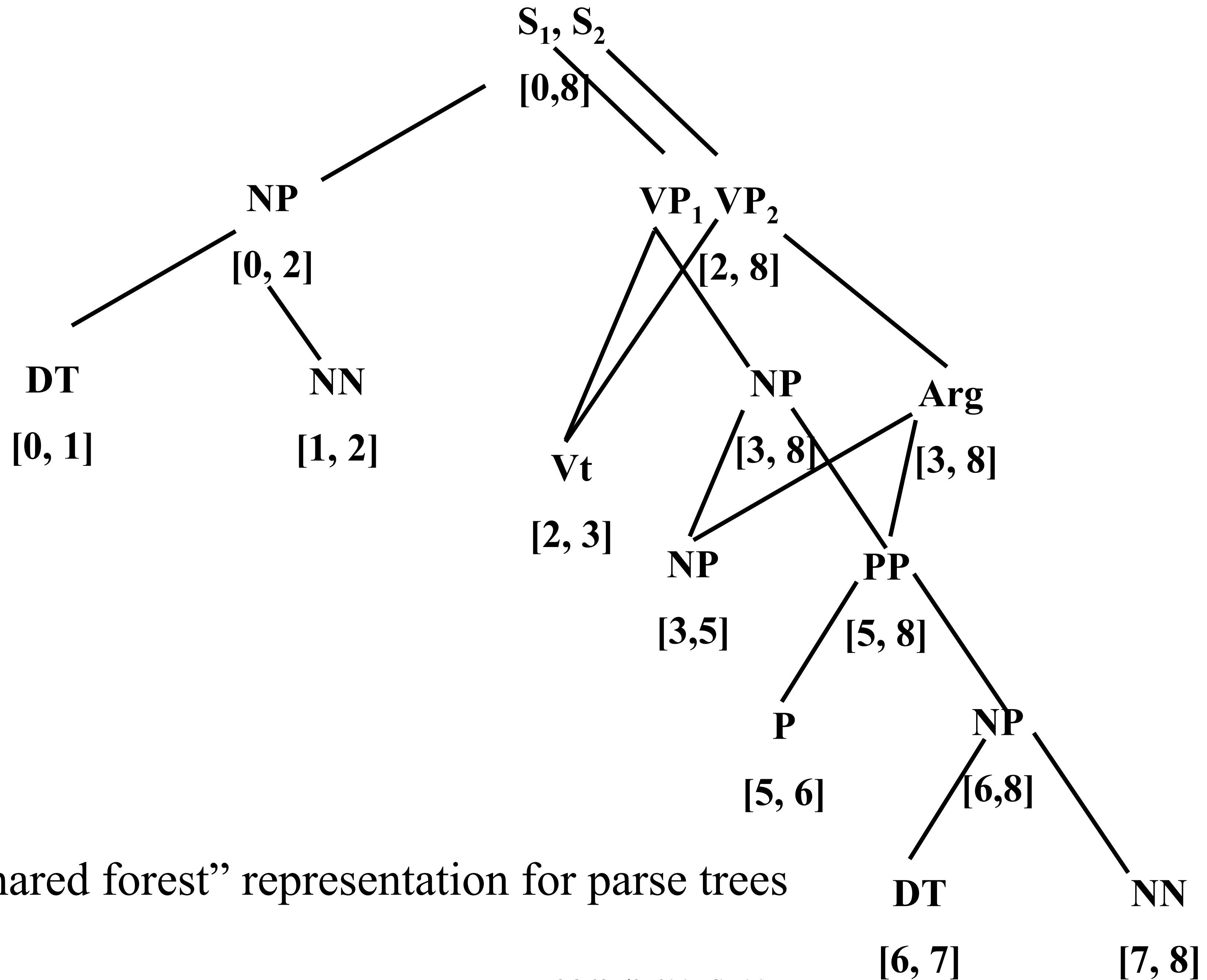
$|G|$

n

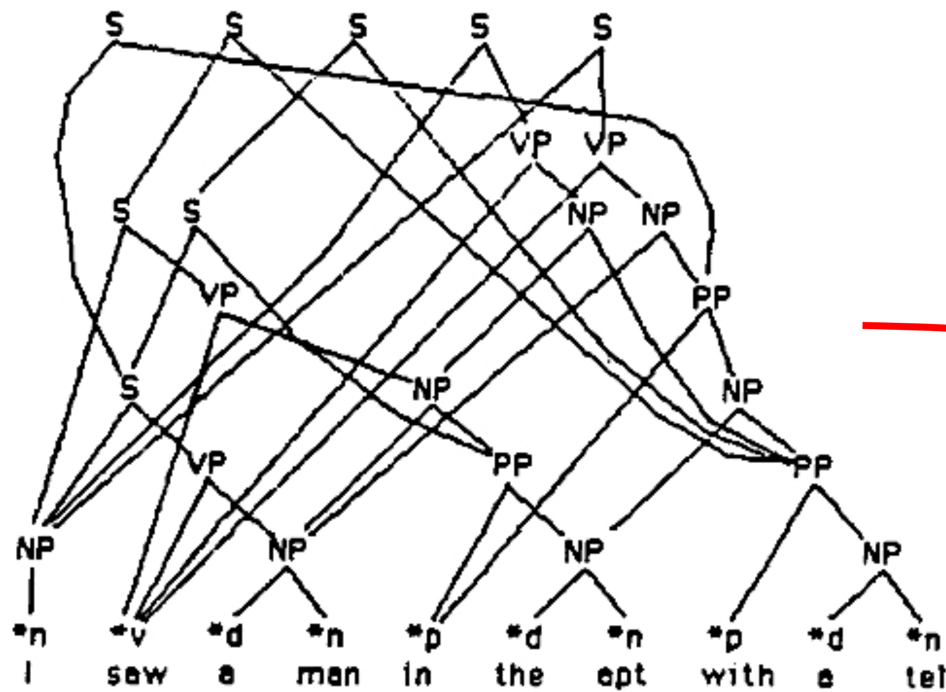
Max time to process any *one* item in a state set

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 .

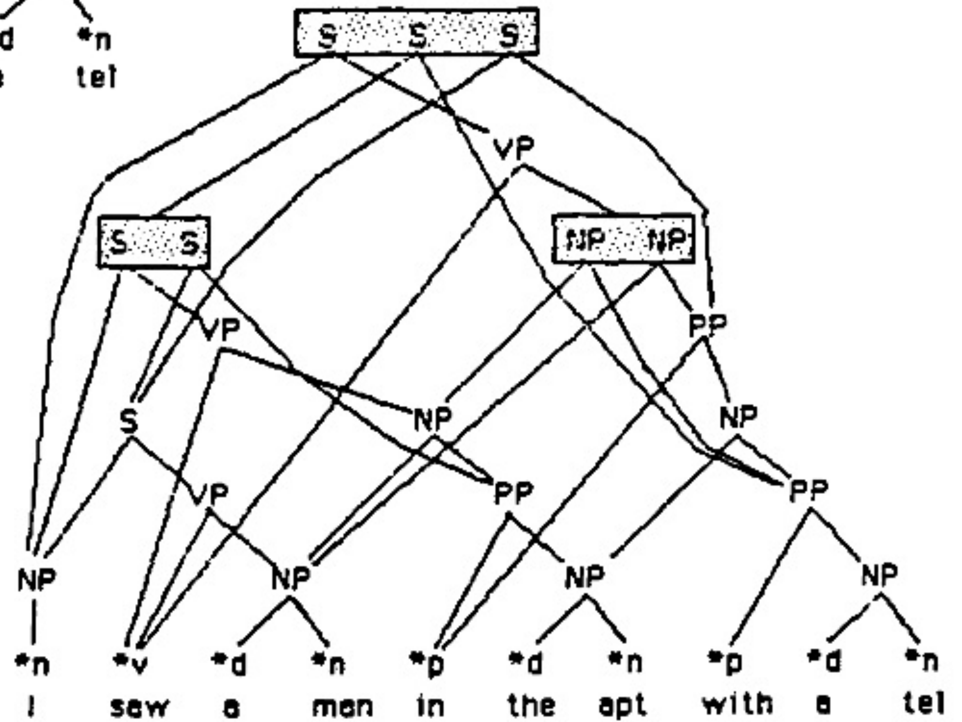
Go back to column *i* & Advance dot in all items in column, to add to current state set, so $O(|G|n)$



"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 <u>must</u> 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 yeat 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, $A \rightarrow B \bullet C ...$) 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 → … rules again.
- Before adding a rule, check it's not a duplicate.
- Slow if there are > 700 VP → … 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

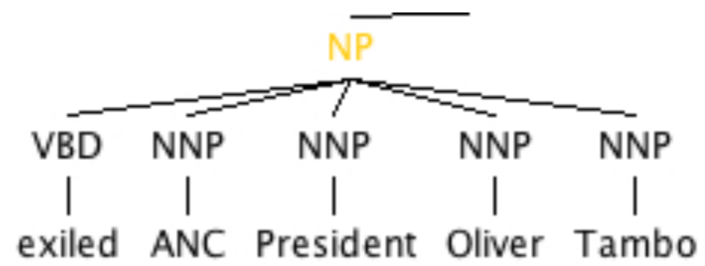
- Consider extracting rules from the 39,000 sentence Wall Street Journal corpus
- There are 694 rules that expand NPs, from

  NP→ PUNC: NN NN NN PUNC.

  to…

  NP → VBN NNS NNS NNS NNS

This is a BIG constant…


NP
VBD
NNP
NNP
NNP
NNP
exiled
ANC
President
Oliver
Tambo

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
	~~1 V . snorted~~
	~~1 P . with~~

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	~~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~~

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	~~0 NP NP . PP~~
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 …