The CYK Algorithm

Informatics 2A: Lecture 20

Adam Lopez

3 November 2016

1. Problems with Parsing as Search
   - The need for ambiguity
   - The computational cost of ambiguity

2. The CYK Algorithm
   - Parsing as intersection
   - Parsing as Dynamic Programming
   - The CYK Algorithm
   - Properties of the Algorithm

Problems with parsers we've learned so far

Deterministic parsing can address grammars with limited ambiguity. For example, LL(1) parsing can handle grammars with no ambiguity.

By re-structuring the grammar, the parser can make a unique decision, based on a limited amount of look-ahead.

Recursive Descent parsing also demands grammar restructuring, in order to eliminate left-recursive rules that can get it into a hopeless loop.

Can we avoid recursion and/ or ambiguity?

Left Recursion

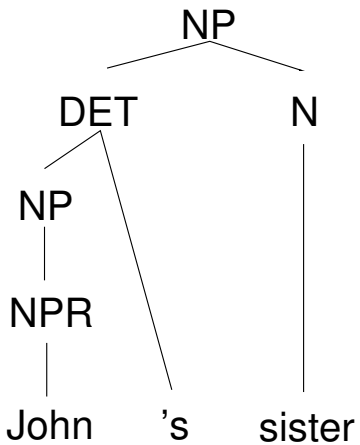
But grammars for natural human languages should be revealing, re-structuring the grammar may destroy this. (Indirectly) left-recursive rules are needed in English.

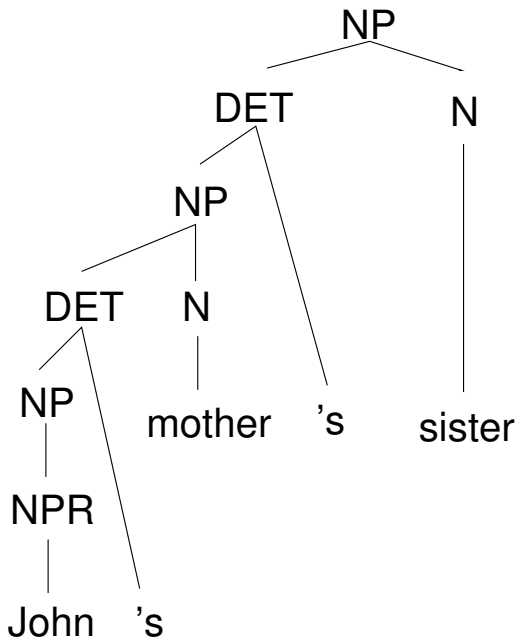
$\text{NP} \rightarrow \text{DET N}$
$\text{NP} \rightarrow \text{NPR}$
$\text{DET} \rightarrow \text{NP 's}$

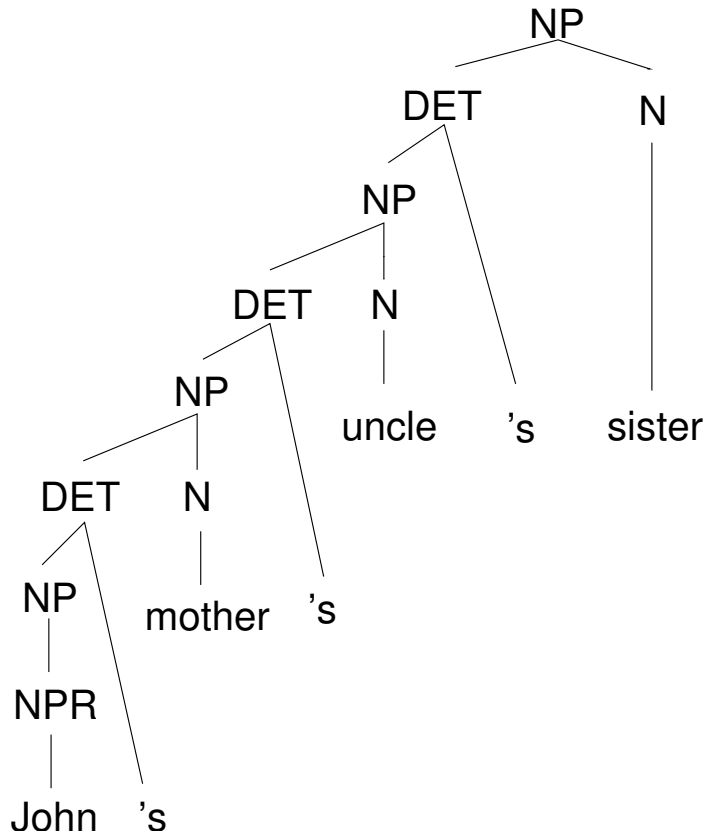
These rules generate NPs with possessive modifiers such as:

John's sister
John's mother's sister
John's mother's uncle's sister
John's mother's uncle's sister's niece

Left Recursion







We don't want to re-structure our grammar rules just to be able to use a particular approach to parsing. Need an alternative.

How necessary is ambiguity?

How necessary is ambiguity?

Can grammars be **inherently ambiguous**?

Suppose we have two languages:
$\mathcal{L}_1 = \{a^n b^n c^m | n, m \geq 0\}$
$\mathcal{L}_2 = \{a^m b^n c^n | n, m \geq 0\}$

Both $\mathcal{L}_1$ and $\mathcal{L}_2$ are context-free. (**Why?**)
The language $\mathcal{L}_1 \cup \mathcal{L}_2$ is also context-free (**Why?**)

In the obvious construction, all strings in the sublanguage $a^n b^n c^n$ have two parses in $\mathcal{L}_1 \cup \mathcal{L}_2$.

It is not possible to create an equivalent grammar with only a single parse for these strings! The proof is very technical, but more approachable with an adaptation of the *context-free pumping lemma* (week 10).

How many parses are there?

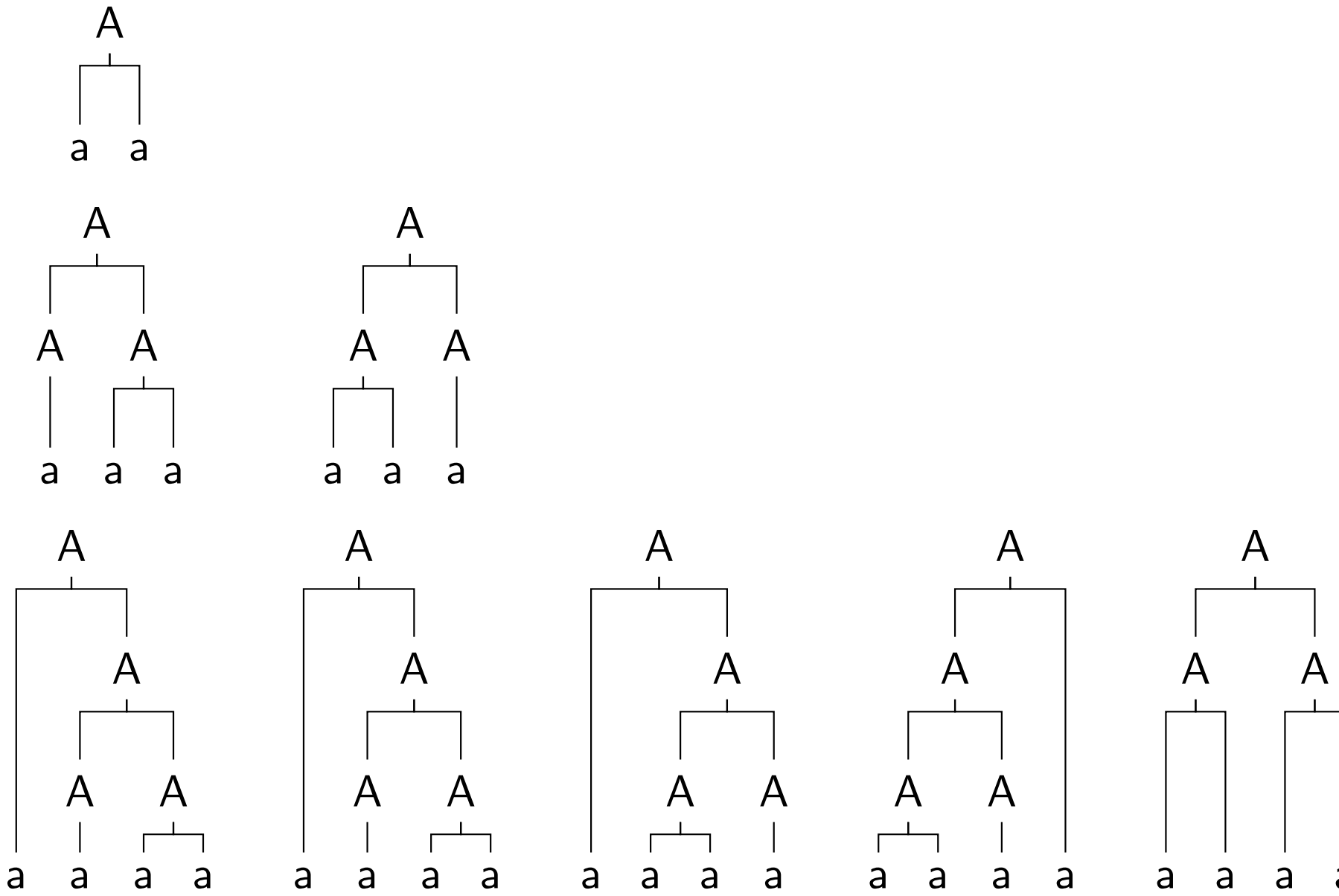
If our grammar is ambiguous (inherently, or by design) then how many possible parses are there?

In general: an infinite number, if we allow unary recursion.

More specific: suppose that we have a grammar in Chomsky normal form. How many possible parses are there for a sentence of n words? Imagine that every nonterminal can rewrite as every pair of nonterminals (A$\rightarrow$BC) and every nonterminal (A$\rightarrow$a)

1. n
2. n^2
3. $n \log n$
4. $\frac{(2n)!}{(n+1)!n!}$

How many parses are there?



How many parses are there?

Intution. Let $C(n)$ be the number of binary trees over a sentence of length n. The root of this tree has two subtrees: one over k words ($1 \leq k < n$), and one over $n - k$ words. Hence, for all values of k, we can combine any subtree over k words with any subtree over $n - k$ words:

$$C(n) = \sum_{k=1}^{n-1} C(k) \times C(n-k)$$

Do several pages of math.

$$C(n) = \frac{(2n)!}{(n+1)!n!}$$

These numbers are called the Catalan numbers. They're big numbers!

n	1	2	3	4	5	6	8	9	10	11	12
$C(n)$	1	1	2	5	14	42	132	429	1430	4862	16796

Problems with Parsing as Search

1. A **recursive descent parser** (top-down) will do badly if there are many different rules for the same LHS. Hopeless for rewriting parts of speech (preterminals) with words (terminals).
2. A **shift-reduce parser** (bottom-up) does a lot of useless work: many phrase structures will be locally possible, but globally impossible. Also inefficient when there is much lexical ambiguity.
3. Both strategies do repeated work by re-analyzing the same substring many times.

We will see how chart parsing solves the re-parsing problem, and also copes well with ambiguity.

Parsing as intersection

Recall: When we worked with FSTs, we could analyse a string by first converting that string to an FSA, and then composing that FSA with the FST that implemented the analyser. Equivalently, we intersect the FSA with the input tape of the FST.

Question. What is the intersection of a regular language and a context-free language?

- A regular language
- A context-free language
- Something else

Claim. The intersection of any regular language and any context-free language is context-free.

Since any CFL can be represented by a CFG and any RL can be represented by a FSA, we will work with languages in those representations. (It's possible to demonstrate this claim using other representations.)

Preliminaries

Recall that every grammar can be converted to Chomsky normal form, in which all rules are of the form A$\rightarrow$BC or A$\rightarrow$w. Suppose that we have a grammar $G = (N, T, P, S)$ of this form, where:

- N is a set of nonterminals
- T is a set of terminals
- P is a set of productions
- $S \in N$ is a start symbol

We also have an ϵ-free FSA $M = (Q, T, \delta, q_0, F)$ where:

- Q is a set of states
- T is an alphabet (note: same as T above)
- $\delta \in Q \times T \times Q$ is a set of transitions
- $q_0 \in Q$ is a start state
- F is a set of final states

Proof Sketch of Claim

We construct a new grammar G'. Its nonterminals are the set $Q \times N \times Q \cup \{S'\}$.

Example. If A is a nonterminal in G, and q, r are states of M, then $\langle q, A, r \rangle$ is a nonterminal in G'.

Add to G' a production $S' \to \langle q_0, S, r \rangle$ for every $r \in F$.

Intution. We will construct G' so that nonterminal $\langle q, A, r \rangle$ derives string w if and only if:

1. A derives w.
2. w is the label of some path from q to r in M.

If we can do this, then we will have that $L(G') = L(G) \cap L(M)$, since S' will derive w if and only if:

1. S derives w.
2. w is the label of a path from q_0 to some state in F.

Proof Sketch of Claim

Step 1. For every production $A \to w$ in P, and every transition $q \to^w r$ in δ, add the production $\langle q, A, r \rangle \to w$ to G'.

By construction, our intuition now holds for any string of length 1.

Step 2. For every production $A \to BC$ in P, and every trio of states q, r, and s in Q, add the production $\langle q, A, r \rangle \to \langle q, B, s \rangle \langle s, C, r \rangle$ to G'.

Assume our intuition holds for any string of length n or smaller. By induction on the form of the rules above, it must also hold on all strings of length $n + 1$. □

Parsing as Intersection

We now have a sketch of an algorithm to parse a sentence with any grammar in CNF:

1. Convert the sentence to a FSA. Recall that if there are n words, then the number of states in this FSA is $n + 1$.
2. Compute the Bar-Hillel construction of the above proof.
3. Check to see if the intersected language is empty. (**How?**)

What is the runtime of this algorithm?

1. $\mathcal{O}(n)$
2. $\mathcal{O}(n^2)$
3. $\mathcal{O}(n^3)$
4. $\mathcal{O}(2^n)$
5. $C(n)$ (the nth Catalan number)

Can we do better?

The algorithm sketched above is inefficient: it computes all possible rules and nonterminals of G', even if they are not used in any parse of the input sentence.

We could do better if we only add rules to the grammar that derive some part of the input.

Fortunately, the algorithm implies (in its inductive step) a way to do this: only add $\langle q, A, r \rangle$ to the grammar if we have already determined that $\langle q, B, s \rangle$ and $\langle s, C, r \rangle$ derive some string of the grammar, for any rule $A \to BC$ and choice of s.

Does this remind you of anything? (It's the same recursion we saw in the Catalan numbers)

Since the base case is at the word level, we can think of this as a bottom-up filter on the construction.

CKY as dynamic programming

We have a Boolean table called Chart, such that $\mathrm{Chart}[A, i, j]$ is true if there is a sub-phrase according the grammar that dominates words i through words j

Build this chart recursively, similarly to the Viterbi algorithm:

Seed the chart:
$\mathrm{Chart}[A, i, i+1] = \mathrm{True}$ if there exists a rule $A \rightarrow w_{i+1}$ where w_{i+1} is the $(i+1)$th word in the string

For $j > i+1$:

$$\mathrm{Chart}[A, i, j] = \bigvee_{k=i+1}^{j-1} \bigvee_{A \rightarrow B\ C} Chart[B, i, k] \wedge Chart[C, k, j]$$

Depicting a Chart

A chart can be depicted as a matrix:

- Rows and columns of the matrix correspond to the start and end positions of a span (ie, starting right before the first word, ending right after the final one);
- A cell in the matrix corresponds to the sub-string that starts at the row index and ends at the column index.
- It can contain information about the type of constituent (or constituents) that span(s) the substring, pointers to its sub-constituents, and/or predictions about what constituents might follow the substring.

CYK Algorithm

The algorithm we have worked out is called the CKY (Cocke, Younger, Kasami) algorithm. It answers the question: given grammar G and string w, is $w \in L(G)$?

- Assumes that the grammar is in Chomsky Normal Form: rules all have form $A \rightarrow BC$ or $A \rightarrow w$.
- Conversion to CNF can be done automatically.

NP	→	Det Nom	NP	→	Det Nom
Nom	→	N | OptAP Nom	Nom	→	*book* | *orange* | AP Nom
OptAP	→	ϵ | OptAdv A	AP	→	*heavy* | *orange* | Adv A
A	→	*heavy* | *orange*	A	→	*heavy* | *orange*
Det	→	*a*	Det	→	*a*
OptAdv	→	ϵ | *very*	Adv	→	*very*
N	→	*book* | *orange*			

CYK: an example

Let's look at a simple example.

Grammar Rules in CNF

NP	$\rightarrow$	Det Nom
Nom	$\rightarrow$	*book* | *orange* | AP Nom
AP	$\rightarrow$	*heavy* | *orange* | Adv A
A	$\rightarrow$	*heavy* | *orange*
Det	$\rightarrow$	*a*
Adv	$\rightarrow$	*very*

(N.B. Converting to CNF sometimes causes duplication!)

Now let's parse: *a very heavy orange book*

Filling out the CYK chart

$_0$ a $_1$ very $_2$ heavy $_3$ orange $_4$ book $_5$

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	NP
1	very		Adv	AP	Nom	Nom
2	heavy			A,AP	Nom	Nom
3	orange				Nom,A,AP	Nom
4	book					Nom

$$\begin{aligned}
\text{NP} &\rightarrow \text{Det Nom} \\
\text{Nom} &\rightarrow \textit{book} \mid \textit{orange} \mid \text{AP Nom} \\
\text{AP} &\rightarrow \textit{heavy} \mid \textit{orange} \mid \text{Adv A} \\
\text{A} &\rightarrow \textit{heavy} \mid \textit{orange} \\
\text{Det} &\rightarrow \textit{a} \\
\text{Adv} &\rightarrow \textit{very}
\end{aligned}$$

CYK: The general algorithm

function CKY-Parse(*words*, *grammar*) **returns** *table* **for**

$j \leftarrow$**from** 1 **to** LENGTH(*words*) **do**
  $table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$
  for $i \leftarrow$**from** $j-2$ **downto** 0 **do**
    for $k \leftarrow i+1$ **to** $j-1$ **do**
      $table[i, j] \leftarrow table[i, j] \cup$
        $\{A \mid A \rightarrow BC \in grammar,$
          $B \in table[i, k]$
          $C \in table[k, j]\}$

CYK: The general algorithm

function CKY-Parse(*words*, *grammar*) **returns** *table* **for**

$j \leftarrow$**from** 1 **to** LENGTH(*words*) **do** — loop over the columns

  $table[j-1, j] \leftarrow \{A \mid A \rightarrow words[j] \in grammar\}$ — fill bottom cell

  for $i \leftarrow$**from** $j-2$ **downto** 0 **do** — fill row i in column j

    for $k \leftarrow i+1$ **to** $j-1$ **do** — loop over split locations between i and j

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

        $\{A \mid A \rightarrow BC \in grammar,$

          $B \in table[i, k]$

          $C \in table[k, j]\}$

Check the grammar for rules that link the constituent in $[i, k]$ with those in $[k, j]$. For each rule found store LHS in cell $[i, j]$.

From CYK Recognizer to CYK Parser

- So far, we just have a chart recognizer, a way of determining whether a string belongs to the given language.
- Changing this to a parser requires recording which existing constituents were combined to make each new constituent.
- This requires another field to record the one or more ways in which a constituent spanning (i,j) can be made from constituents spanning (i,k) and (k,j). (More clearly displayed in graph representation, see next lecture.)
- In any case, for a fixed grammar, the CYK algorithm runs in time $O(n^3)$ on an input string of n tokens.
- The algorithm identifies all possible parses.

CYK-style parse charts

Even without converting a grammar to CNF, we can draw *CYK-style* parse charts:

		1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0	a	Det			NP	NP
1	very		OptAdv	OptAP	Nom	Nom
2	heavy			A,OptAP	Nom	Nom
3	orange				N,Nom,A,AP	Nom
4	book					N,Nom

(We haven't attempted to show ϵ-phrases here. Could in principle use cells below the main diagonal for this . . .)
However, CYK-style parsing will have run-time worse than $O(n^3)$ if e.g. the grammar has rules $A \rightarrow BCD$. (**Exercise:** Why?)

Second example

Grammar Rules in CNF

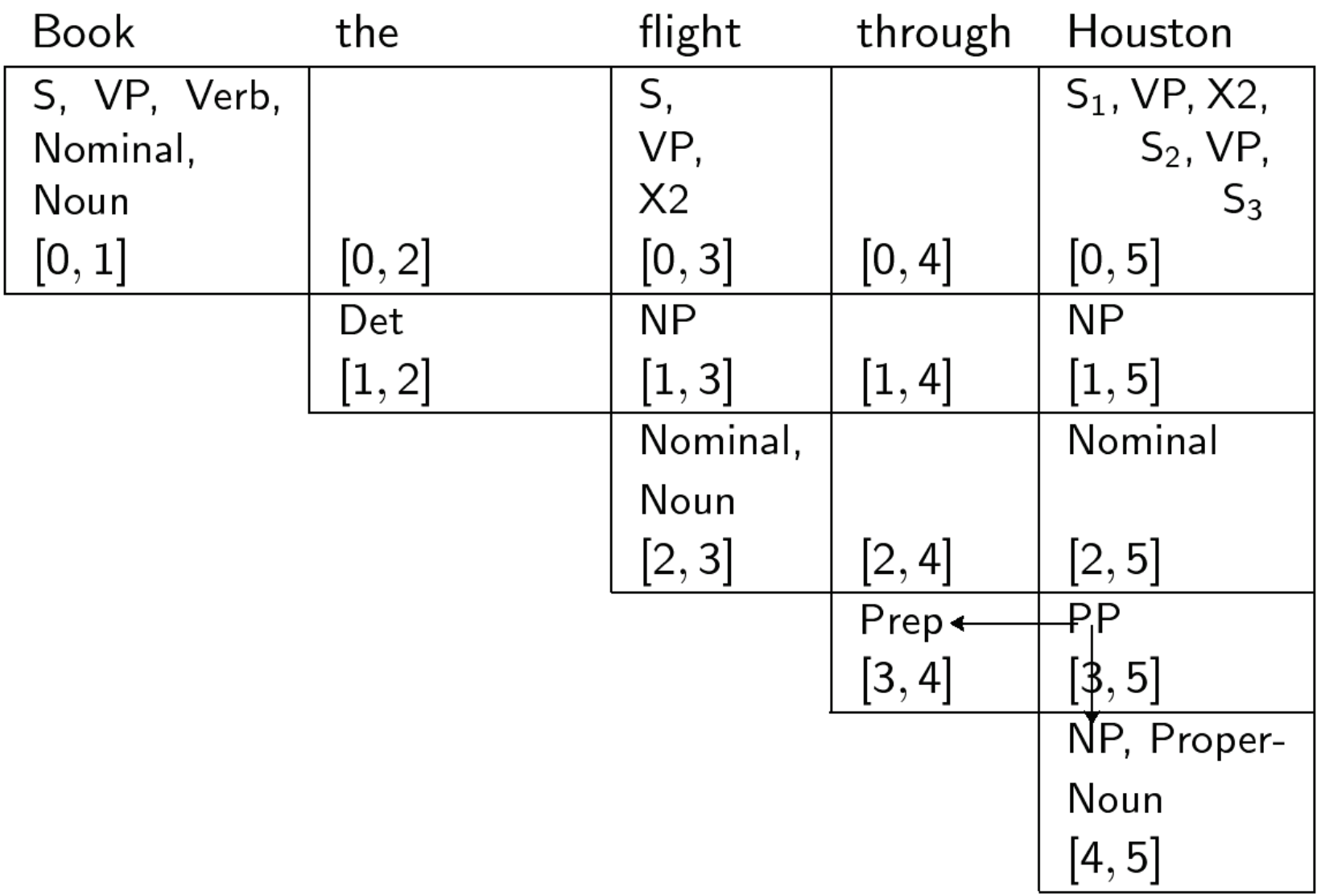
$S \rightarrow NP\ VP$	$Nominal \rightarrow book \mid flight \mid money$
$S \rightarrow X1\ VP$	$Nominal \rightarrow Nominal\ noun$
$X1 \rightarrow Aux\ VP$	$Nominal \rightarrow Nominal\ PP$
$S \rightarrow book \mid include \mid prefer$	$VP \rightarrow book \mid include \mid prefer$
$S \rightarrow Verb\ NP$	$VPVerb \rightarrow NP$
$S \rightarrow X2$	$VP \rightarrow X2\ PP$
$S \rightarrow Verb\ PP$	$X2 \rightarrow Verb\ NP$
$S \rightarrow VP\ PP$	$VP \rightarrow Verb\ NP$
$NP \rightarrow TWA \mid Houston$	$VP \rightarrow VP\ PP$
$NP \rightarrow Det\ Nominal$	$PP \rightarrow Preposition\ NP$
$Verb \rightarrow book \mid include \mid prefer$	$Noun \rightarrow book \mid flight \mid money$

Let's parse *Book the flight through Houston*!

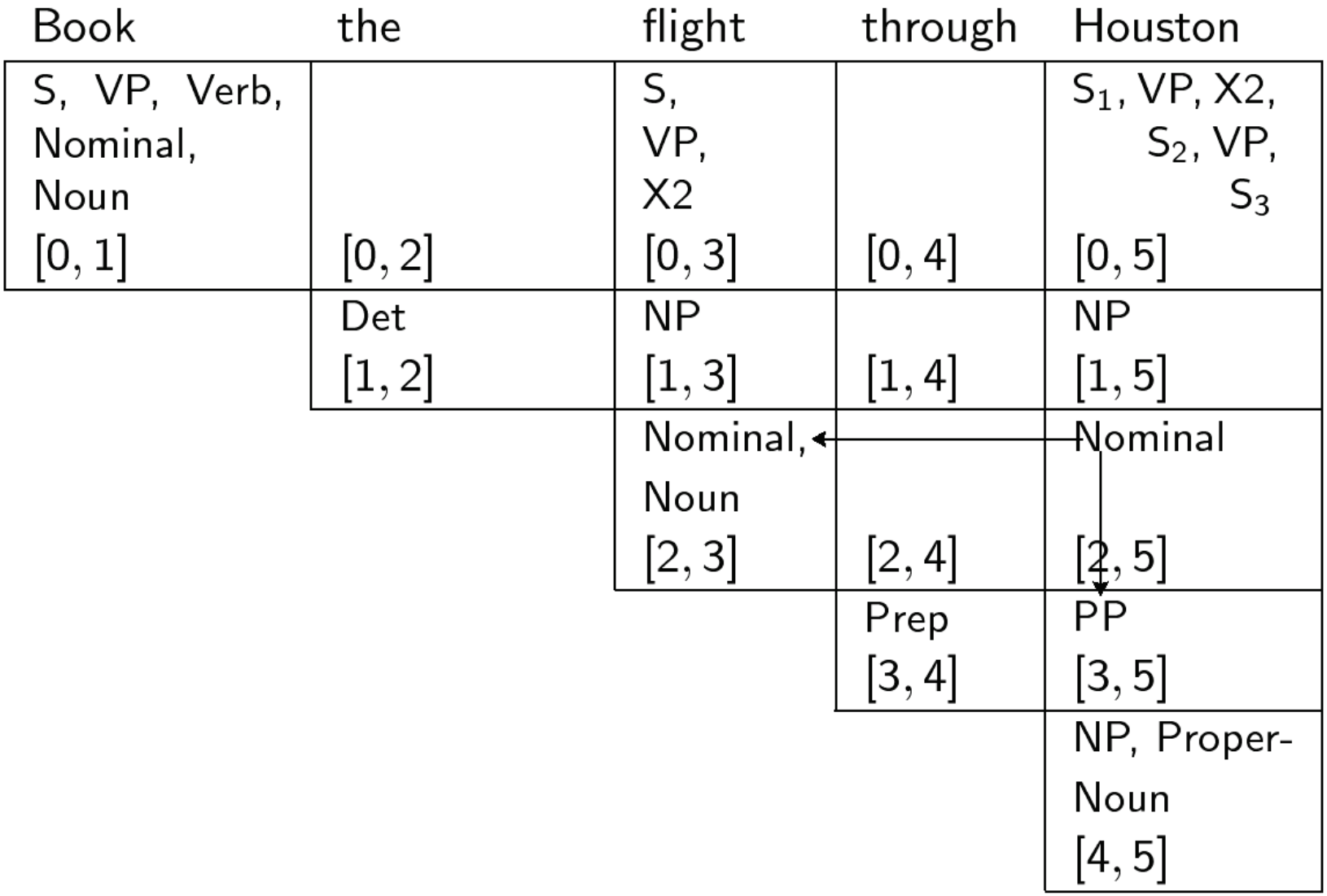
Second example

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]	[0, 4]	S_1, VP, X2, S_2, VP, S_3 [0, 5]
	Det [1, 2]	NP [1, 3]	[1, 4]	NP [1, 5]
		Nominal, Noun [2, 3]	[2, 4]	Nominal [2, 5]
			Prep [3, 4]	PP [3, 5]
				NP, Proper-Noun [4, 5]

Visualizing the Chart



Visualizing the Chart



Dynamic Programming as a problem-solving technique

- Given a problem, systematically fill a table of solutions to sub-problems: this is called memoization.
- Once solutions to all sub-problems have been accumulated, solve the overall problem by composing them.
- For parsing, the sub-problems are analyses of sub-strings and correspond to constituents that have been found.
- Sub-trees are stored in a chart (aka well-formed substring table), which is a record of all the substructures that have ever been built during the parse.

Solves **re-parsing problem**: sub-trees are looked up, not re-parsed!
Solves **ambiguity problem**: chart implicitly stores all parses!

Summary

- Parsing as search is inefficient (typically exponential time).
- Alternative: use dynamic programming and memoize sub-analysis in a chart to avoid duplicate work.
- The chart can be visualized as as a matrix.
- The CYK algorithm builds a chart in $O(n^3)$ time. The basic version gives just a recognizer, but it can be made into a parser if more info is recorded in the chart.

Reading: J&M (2nd ed), Chapter. 13, Sections 13.3–13.4
NLTK Book, Chapter. 8 (*Analyzing Sentence Structure*), Section 8.4

Next lecture: the Earley parser or dynamic programming for top-down parsing