Recursive Descent Parsing and CYK

ANLP: Lecture 13

Shay Cohen

14 October 2019

Last Class

Chomsky normal form grammars

English syntax

Agreement phenomena and the way to model them with CFGs

Recap: Syntax

Two reasons to care about syntactic structure (parse tree):

- As a guide to the semantic interpretation of the sentence
- As a way to prove whether a sentence is grammatical or not

But having a grammar isn't enough.

We also need a parsing algorithm to compute the parse tree for a given input string and grammar.

Parsing algorithms

Goal: compute the structure(s) for an input string given a grammar.

- As usual, ambiguity is a huge problem.
  - For correctness: need to find the right structure to get the right meaning.
  - For efficiency: searching all possible structures can be very slow; want to use parsing for large-scale language tasks (e.g., used to create Google's "infoboxes").

Global and local ambiguity

- We've already seen examples of global ambiguity: multiple analyses for a full sentence, like I saw the man with the telescope
- But local ambiguity is also a big problem: multiple analyses for parts of sentence.
  - the dog bit the child: first three words could be NP (but aren't).
  - Building useless partial structures wastes time.
  - Avoiding useless computation is a major issue in parsing.
- Syntactic ambiguity is rampant; humans usually don't even notice because we are good at using context/semantics to disambiguate.

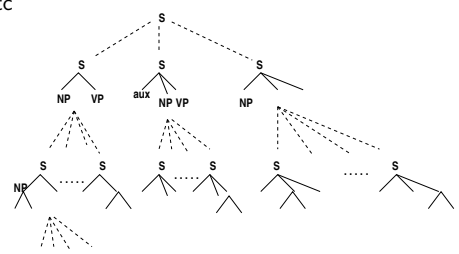
Parser properties

All parsers have two fundamental properties:

- Directionality: the sequence in which the structures are constructed.
  - top-down: start with root category (S), choose expansions, build down to words.
  - bottom-up: build subtrees over words, build up to S.
  - Mixed strategies also possible (e.g., left corner parsers)
- Search strategy: the order in which the search space of possible analyses is explored.

Example: search space for top-down parser

- Start with S node.
- Choose one of many possible expansions.
- Each of which has children with many possible expansions...
- etc



Search strategies

- depth-first search: explore one branch of the search space at a time, as far as possible. If this branch is a dead-end, parser needs to backtrack.
- breadth-first search: expand all possible branches in parallel (or simulated parallel). Requires storing many incomplete parses in memory at once.
- best-first search: score each partial parse and pursue the highest-scoring options first. (Will get back to this when discussing statistical parsing.)

Recursive Descent Parsing

- A recursive descent parser treats a grammar as a specification of how to break down a top-level goal (find `S`) into subgoals (find `NP VP`).
- It is a **top-down**, **depth-first** parser:
  - Blindly expand nonterminals until reaching a terminal (word).
  - If multiple options available, choose one but store current state as a backtrack point (in a **stack** to ensure depth-first.)
  - If terminal matches next input word, continue; else, backtrack.

RD Parsing algorithm

Start with subgoal = `S`, then repeat until input/subgoals are empty:

- If first subgoal in list is a **non-terminal** `A`, then pick an expansion `A` → `B C` from grammar and replace `A` in subgoal list with `B C`
- If first subgoal in list is a **terminal** `w`:
  - If input is empty, backtrack.
  - If next input word is different from `w`, backtrack.
  - If next input word is `w`, match! i.e., consume input word `w` and subgoal `w` and move to next subgoal.

If we run out of backtrack points but not input, no parse is possible.

Recursive descent parsing pseudocode

In the background: a CFG G, a sentence $x_1 \cdots x_n$
Function RecursiveDescent(t, v, i) where

- t is a partially constructed tree
- v is a node in t
- i is a sentence position
- Let N be the nonterminal in v
- For each rule with LHS N:
  - If the rule is a lexical rule $N \to w$, check whether $x_i = w$, if so increase i by 1 and call RecursiveDescent(t, u, $i+1$) where u is the lowest point above v that has a nonterminal
  - If the rule is a grammatical rule, Let t' be t with v expanded using the rule $N \to A_1 \cdots A_n$. For each $j \in \{1 \cdots n\}$, call RecursiveDescent(t', u_j, i) where u_j is the node for nonterminal A_j in t'.

Start with: RecursiveDescent(S, $\mathrm{topnode}$, 1)

Quick quiz: this algorithm has a bug. Where? What do we need to add?

Recursive descent example

Consider a very simple example:

- Grammar contains only these rules:

`S` → `NP VP`	`VP` → `V`	`NN` → `bit`	`V` → `bit`
`NP` → `DT NN`	`DT` → `the`	`NN` → `dog`	`V` → `dog`

- The input sequence is the dog bit

Recursive descent example

- Operations:
  - Expand (E)
  - Match (M)
  - Backtrack to step n (Bn)

Step	Op.	Subgoals	Input
0		S	the dog bit
1	E	NP VP	the dog bit
2	E	DT NN VP	the dog bit
3	E	the NN VP	the dog bit
4	M	NN VP	dog bit
5	E	bit VP	dog bit
6	B4	NN VP	dog bit
7	E	dog VP	dog bit
8	M	VP	bit
9	E	V	bit
10	E	bit	bit
11	M		

Recursive Descent Parsing

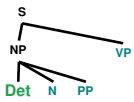
S

the dog saw a man in the park

Recursive Descent Parsing

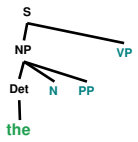
the dog saw a man in the park

Recursive Descent Parsing



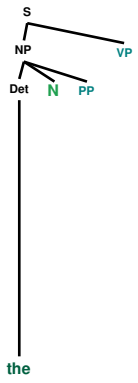
the dog saw a man in the park

Recursive Descent Parsing



the dog saw a man in the park

Recursive Descent Parsing



the dog saw a man in the park

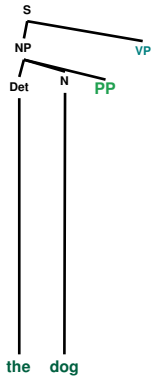
Recursive Descent Parsing

the dog saw a man in the park

Recursive Descent Parsing

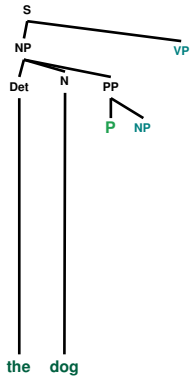
the dog saw a man in the park

Recursive Descent Parsing



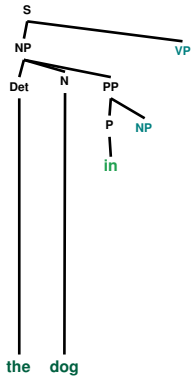
the dog saw a man in the park

Recursive Descent Parsing



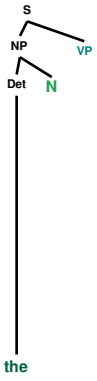
the dog saw a man in the park

Recursive Descent Parsing



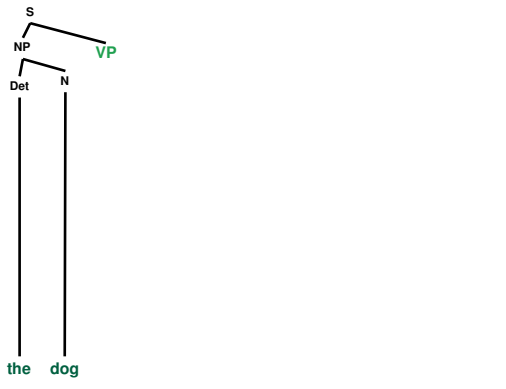
the dog saw a man in the park

Recursive Descent Parsing



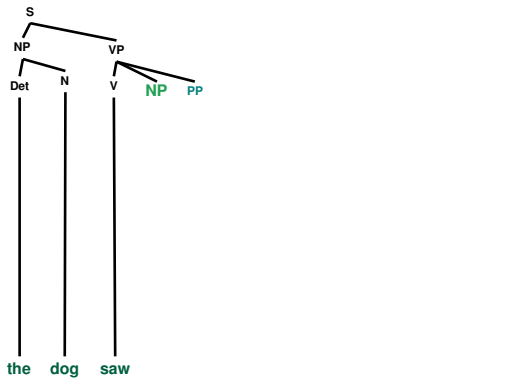
the dog saw a man in the park

Recursive Descent Parsing



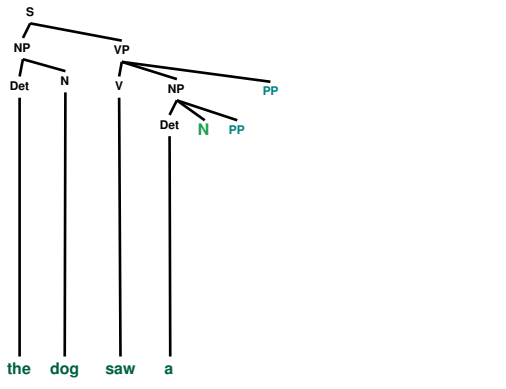
the dog saw a man in the park

Recursive Descent Parsing



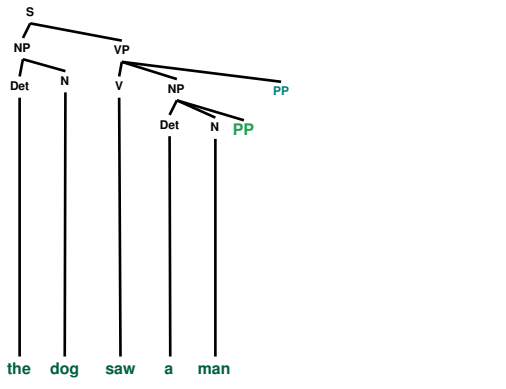
the dog saw a man in the park

Recursive Descent Parsing



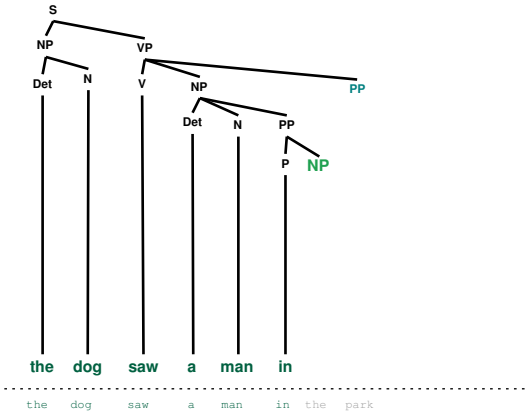
the dog saw a man in the park

Recursive Descent Parsing

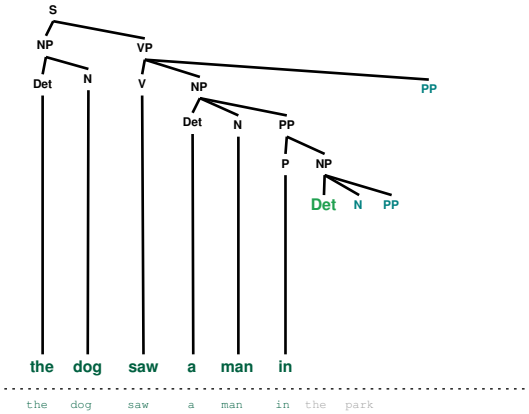


the dog saw a man in the park

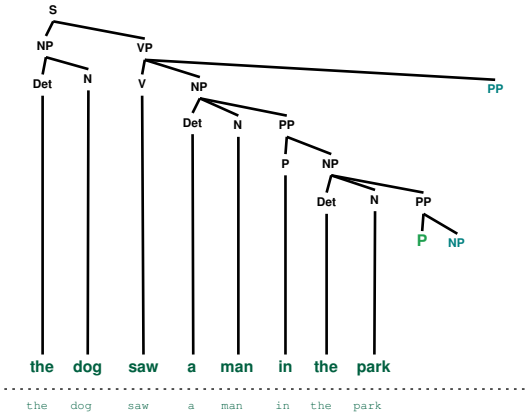
Recursive Descent Parsing



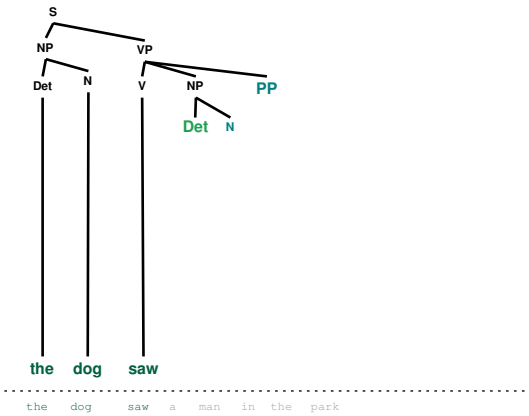
Recursive Descent Parsing



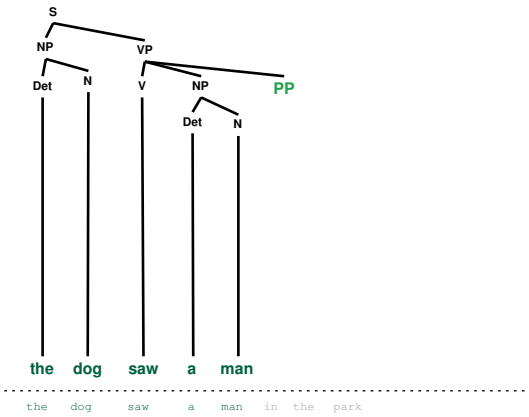
Recursive Descent Parsing



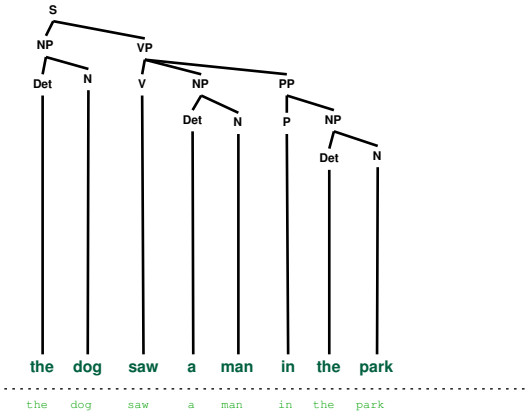
Recursive Descent Parsing



Recursive Descent Parsing



Recursive Descent Parsing



Left Recursion

Can recursive descent parsing handle left recursion?
Grammars for natural human languages should be revealing, left-recursive rules are needed in English.

NP → DET N
NP → NPR
DET → 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

Shift-Reduce Parsing

A Shift-Reduce parser tries to find sequences of words and phrases that correspond to the righthand side of a grammar production and replace them with the lefthand side:

- **Directionality** = bottom-up: starts with the words of the input and tries to build trees from the words up.
- **Search strategy** = breadth-first: starts with the words, then applies rules with matching right hand sides, and so on until the whole sentence is reduced to an S.

Algorithm Sketch: Shift-Reduce Parsing

Until the words in the sentences are substituted with S:

- Scan through the input until we recognise something that corresponds to the RHS of one of the production rules (shift)
- Apply a production rule in reverse; i.e., replace the RHS of the rule which appears in the sentential form with the LHS of the rule (reduce)

A shift-reduce parser implemented using a stack:

1. start with an empty stack
2. a shift action pushes the current input symbol onto the stack
3. a reduce action replaces n items with a single item

Shift-Reduce Parsing

Stack	Remaining Text
	my dog saw a man in the park

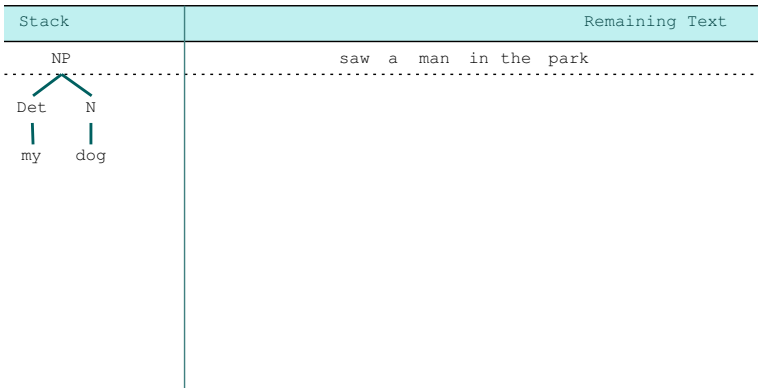
Shift-Reduce Parsing

Stack	Remaining Text
Det	dog saw a man in the park
my	

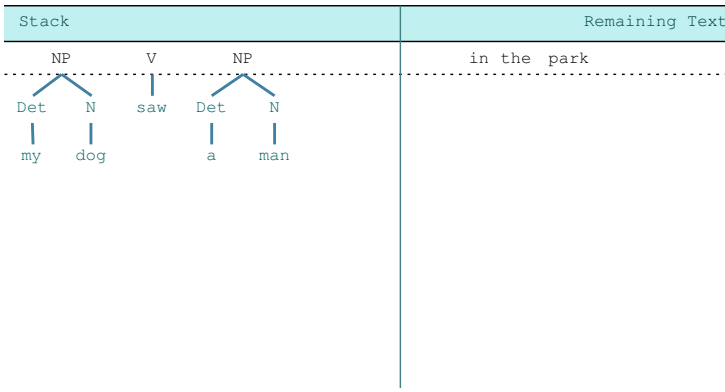
Shift-Reduce Parsing

Stack	Remaining Text
Det N	saw a man in the park
my dog	

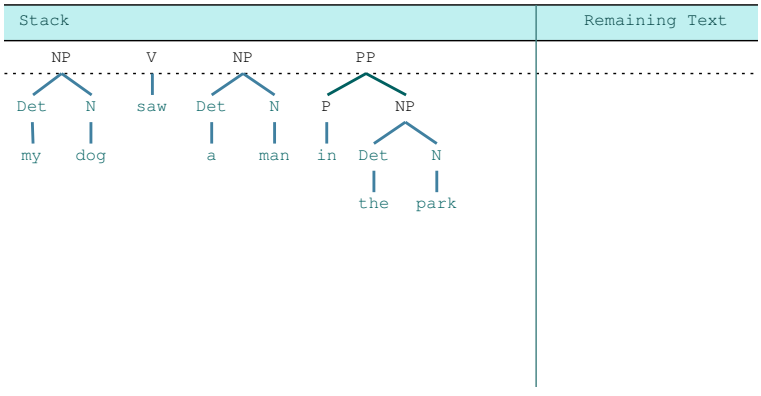
Shift-Reduce Parsing



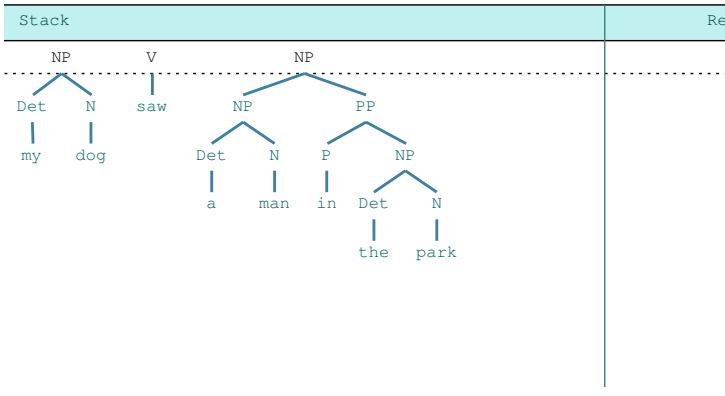
Shift-Reduce Parsing



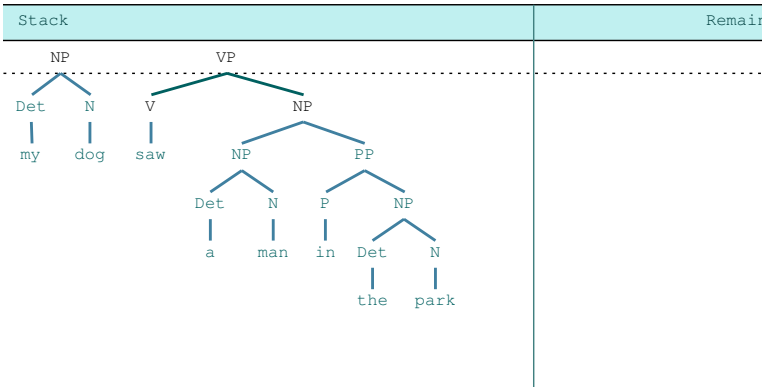
Shift-Reduce Parsing



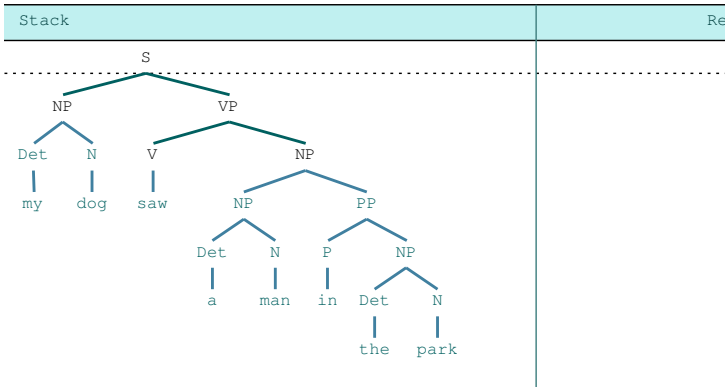
Shift-Reduce Parsing



Shift-Reduce Parsing



Shift-Reduce Parsing



How many parses are there?

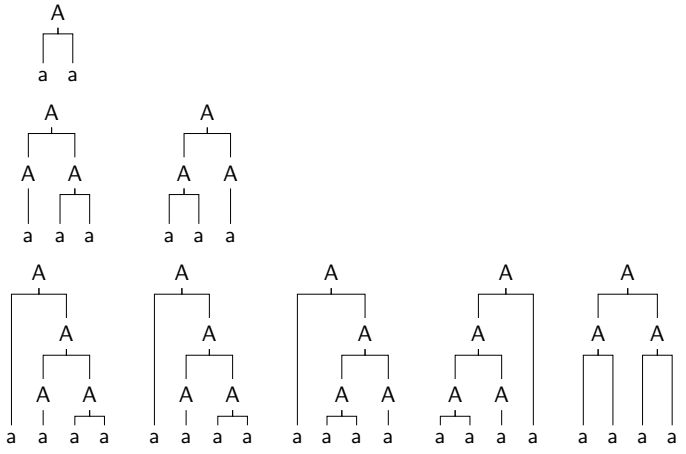
If our grammar is ambiguous (inherently, or by design) then how many possible parses are there?

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→BC) and every nonterminal (A→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:

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

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

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

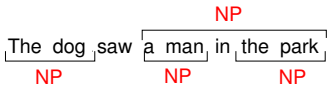
Problems with Parsing as Search

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.

Dynamic Programming

With a CFG, a parser should be able to avoid re-analyzing sub-strings because the analysis of any sub-string is independent of the rest of the parse.



The parser's exploration of its search space can exploit this independence if the parser uses dynamic programming.

Dynamic programming is the basis for all chart parsing algorithms.

Parsing as Dynamic Programming

- 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!

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

CYK (Cocke, Younger, Kasami) is an algorithm for recognizing and recording constituents in the chart.

- 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 before we explain the general case.

Grammar Rules in CNF

NP	→	Det Nom
Nom	→	*book* | *orange* | AP Nom
AP	→	*heavy* | *orange* | Adv A
A	→	*heavy* | *orange*
Det	→	*a*
Adv	→	*very*

(N.B. Converting to CNF sometimes breeds 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	2	3	4	5
		a	*very*	*heavy*	*orange*	*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

CYK: The general algorithm

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

j ←**from** 1 **to** LENGTH(*words*) **do**
  table[*j* − 1, *j*] ← {*A* | *A* → *words*[*j*] ∈ *grammar*}
  for *i* ←**from** *j* − 2 **downto** 0 **do**
    for *k* ← *i* + 1 **to** *j* − 1 **do**
      table[*i*, *j*] ← *table*[*i*, *j*]∪
        {*A* | *A* → *BC* ∈ *grammar*,
          B ∈ *table*[*i*, *k*]
          C ∈ *table*[*k*, *j*]}

CYK: The general algorithm

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

j ←**from** 1 **to** LENGTH(*words*) **do** — loop over the columns
  table[*j* − 1, *j*] ← {*A* | *A* → *words*[*j*] ∈ *grammar*} — fill bottom cell
  for *i* ←**from** *j* − 2 **downto** 0 **do** — fill row *i* in column *j*
    for *k* ← *i* + 1 **to** *j* − 1 **do** — loop over split locations between *i* and *j*
      table[*i*, *j*] ← *table*[*i*, *j*]∪
        {*A* | *A* → *BC* ∈ *grammar*, — Check the grammar for rules that link the constituents in [*i*, *k*] with those in [*k*, *j*]. For each rule found store LHS in cell [*i*, *j*].
          B ∈ *table*[*i*, *k*]
          C ∈ *table*[*k*, *j*]}

A succinct representation of CKY

We have a Boolean table called Chart, such that 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:

For $j > i + 1$:

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

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

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 $A \rightarrow BCD$.

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*!

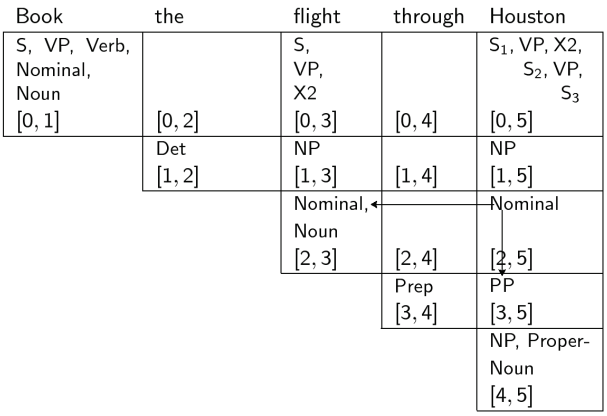
Second example

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

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



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!

A Tribute to CKY (part 1/3)

You, my CKY algorithm,
dictate every parser's rhythm,
if Cocke, Younger and Kasami hadn't bothered,
all of our parsing dreams would have been shattered.

You are so simple, yet so powerful,
and with the proper semiring and time,
you will be truthful,
to return the best parse - anything less would be a crime.

With dynamic programming or memoization,
you are one of a kind,
I really don't need to mention,
if it weren't for you, all syntax trees would be behind.

A Tribute to CKY (part 2/3)

Failed attempts have been made to show there are better,
for example, by using matrix multiplication,
all of these impractical algorithms didn't matter –
you came out stronger, insisting on just using summation.

All parsing algorithms to you hail,
at least those with backbones which are context-free,
you will never become stale,
as long as we need to have a syntax tree.

It doesn't matter that the C is always in front,
or that the K and Y can swap,
you are still on the same hunt,
maximizing and summing, nonstop.

A Tribute to CKY (part 3/3)

Every Informatics student knows you intimately,
they have seen your variants dozens of times,
you have earned that respect legitimately,
and you will follow them through their primes.

CKY, going backward and forward,
inside and out,
it is so straightforward -
You are the best, there is no doubt.

Questions to Ask Yourself

- How many spans are there for a given sequence (as a function of the length of the sentence)?
- How long does it take to process each one of them (each "cell") as a function of the length of the span and the size of the grammar?
- Does CYK perform any unnecessary calculation?

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.