Constituency Parsing, Part I:
The \texttt{C\{KY,YK\}} algorithm

The diagram shows a constituency parse tree and a corresponding chart for the sentence: "the teacher gave the lecture."
Game plan for today:

Some unfinished business from last time...

CYK recognition, step by step

Going from recognition to parsing
“One morning, I shot an elephant in my pajamas. How it got into my pajamas, I don’t know.”
“One morning, I shot an elephant in my pajamas. How it got into my pajamas, I don’t know.”
I shot an elephant in my pajamas.
I shot an elephant in my pajamas.
Ambiguity like this is the bane of parsing...

... and much work goes into dealing with it.

Different grammar design, using probabilities, classifiers, etc. etc.
So: how do we get these trees (ambiguous though they may be) in the first place, anyway?
There are two general approaches to parsing: “top-down”, and “bottom-up.”

“Top-down” parsing starts at the top of the tree, and tries combinations of productions until it gets to the end.

“Bottom-up” parsing does the opposite, and starts with the words themselves and works upwards.
The CYK algorithm is bottom-up, but uses dynamic programming to avoid duplicating effort.

It was independently discovered (!) by John Cocke, Daniel Younger and Tadao Kasami.*

*All of whom had pre-Internet careers, and as such do not have easily-findable photos.
A grammar $G = (N, \Sigma, R, S)$ is in CNF if all productions in $R$ are in one of two forms:

- $A \rightarrow B C$ s.t. $A, B, \text{ and } C \in N$ (all are non-terminals)
- $A \rightarrow a$ s.t. $A \in N \text{ and } a \in \Sigma$ (unary nonterm-term production)

CYK will work on any CFG, as long as it is in Chomsky Normal Form.
The red teacher gave the lecture.
The basic data structure used by CYK is the “chart”...

... whose job it is to hold all possible ways to produce a given string from a CFG.

By looking at how the chart gets filled in, we can recognize strings from a grammar, as well as recover parse trees.
Caution!

J&M use a different chart orientation, and different notation, than just about everybody else in the world.

As such, other lecture slides, websites, etc. may not be as useful as one might like.
Caution!

This is especially important with CYK, since there are so many ways to encounter off-by-one errors.

Remember: the important thing is what the indexes point to, not what their numbers are.
the teacher gave the lecture
Overview of the algorithm:

For each word of input...

  Initialize that word’s terminal cell, then...
  Fill in that word’s column of the chart by...

    Looking at each cell in the column, and finding rules from the grammar that could produce a constituent at that location in the chart.

    When you find a rule, store its LHS in the current cell.

We’ll walk through this with our toy example.
the teacher gave the lecture
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
the teacher gave the lecture
the teacher gave the lecture
The teacher gave the lecture.
the teacher gave the lecture
The teacher gave the lecture.
the teacher gave the lecture
The teacher gave the lecture.
The teacher gave the lecture.
the teacher gave the lecture
the teacher gave the lecture
the teacher gave the lecture
the teacher gave the lecture
the teacher gave the lecture
<table>
<thead>
<tr>
<th></th>
<th>the</th>
<th>teacher</th>
<th>gave</th>
<th>the</th>
<th>lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DT</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>VB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>DT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
VP → VB NP
DT → the
NN → teacher
NN → lecture
VB → gave
the teacher gave the lecture
the teacher gave the lecture
The teacher gave the lecture.
The teacher gave the lecture.
the teacher gave the lecture
the teacher gave the lecture
the teacher gave the lecture
The teacher gave the lecture.

**Diagram:**

The diagram shows the syntactic structure of the sentence. Each node represents a part of speech, and the arrows indicate the grammatical relationships between the parts.

- **0:** `DT` (determiner) → `NP` (noun phrase)
- **1:** `NN` (noun) → `NP` (noun phrase)
- **2:** `VB` (verb) → `NP` (noun phrase)
- **3:** `DT` (determiner)
- **4:** `NN` (noun) → `the` (determiner)
- **5:** `NN` (noun) → `teacher`

**Grammar Rules:**

- `S` → `NP VP`
- `NP` → `DT NN`
- `VP` → `VB NP`

**POST:**

- `S` → `NP VP`
- `NP` → `DT NN`
- `VP` → `VB NP`
- `DT` → `the`
- `NN` → `teacher`
- `NN` → `lecture`
- `VB` → `gave`
The teacher gave the lecture.

S → NP VP
NP → DT NN
VP → VB NP
DT → the
NN → teacher
NN → lecture
VB → gave
the teacher gave the lecture
The teacher gave the lecture.
The teacher gave the lecture.
**Textual Representation:**

The teacher gave the lecture.

**Diagram:**

```
  the  teacher  gave  the  lecture

0  DT  NP  

1  NN  

2  VB  

3  DT  

4  NN  

S → NP VP
NP → DT NN
VP → VB NP
DT → the
NN → teacher
NN → lecture
VB → gave
```
The teacher gave the lecture.
The teacher gave the lecture.
the teacher gave the lecture
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
the teacher gave the lecture

<table>
<thead>
<tr>
<th></th>
<th>the</th>
<th>teacher</th>
<th>gave</th>
<th>the</th>
<th>lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DT</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>VB</td>
<td>VP</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>DT</td>
<td>NP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
VP → VB NP
DT → the
NN → teacher
NN → lecture
VB → gave
The teacher gave the lecture.
the teacher gave the lecture

S → NP VP
NP → DT NN
VP → VB NP
DT → the
NN → teacher
NN → lecture
VB → gave
The teacher gave the lecture.
The teacher gave the lecture.

**Parsing Rules:**

- **S → NP VP**
- **NP → DT NN**
- **VP → VB NP**
- **DT → the**
- **NN → teacher**
- **NN → lecture**
- **VB → gave**
The teacher gave the lecture.
The teacher gave the lecture.
the teacher gave the lecture
The teacher gave the lecture.

```
<table>
<thead>
<tr>
<th></th>
<th>the</th>
<th>teacher</th>
<th>gave</th>
<th>the</th>
<th>lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DT</td>
<td>NP</td>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>VB</td>
<td>VP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>DT</td>
<td>NP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>NN</td>
<td></td>
</tr>
</tbody>
</table>
```

```
S → NP VP
NP → DT NN
VP → VB NP
DT → the
NN → teacher
NN → lecture
VB → gave
```
the teacher gave the lecture

<table>
<thead>
<tr>
<th></th>
<th>the</th>
<th>teacher</th>
<th>gave</th>
<th>the</th>
<th>lecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DT</td>
<td>NP</td>
<td></td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>VB</td>
<td>VP</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>DT</td>
<td>NP</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NN</td>
</tr>
</tbody>
</table>

S → NP VP
NP → DT NN
VP → VB NP
DT → the
NN → teacher
NN → lecture
VB → gave
This algorithm works as a *recognizer*, just as an FSA does for a regular language.

If the \([0, N]\) cell has a production in it, the string came from the language specified by the CFG.

What do we need to do to recover the syntax tree?
The trick is to store *backpointers* for each cell's productions, so we can know where each one came from.

This will allow us to recover the tree...
the teacher gave the lecture
the teacher gave the lecture
The teacher gave the lecture.
the teacher gave the lecture
The teacher gave the lecture.
To recover the tree, we simply recursively follow the backpointers, in pre-order.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
the teacher gave the lecture
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
the teacher gave the lecture
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
The teacher gave the lecture.
Note: In the “real world”, cells would contain many possible productions...

... and this parse-extraction method will only return one of them.

Enumerating them all could be expensive, and is usually not what we actually want.

The solution? Probabilities!
Extending CYK to use a PCFG:

For each word of input...
  Initialize that word’s terminal cell, then...
  Fill in that word’s column of the chart by...
    Looking at each cell in the column, and finding rules from the grammar that could produce a constituent at that location in the chart.

  When you find a rule, store its LHS in the current cell.
Extending CYK to use a PCFG:

For each word of input...

Initialize that word’s terminal cell, then...

Fill in that word’s column of the chart by...

Looking at each cell in the column, and finding rules from the grammar that could produce a constituent at that location in the chart.

When you find a rule, store its LHS in the current cell.
Extending CYK to use a PCFG:

For each word of input...

  Initialize that word’s terminal cell, then...

  Fill in that word’s column of the chart by...

    Looking at each cell in the column, and finding rules from the grammar that could produce a constituent at that location in the chart.

    When you find a rule, store the product of its probability in the PCFG with the probabilities in the two cells that gave rise to it.
the - DT

teacher - NN
gave
the
lecture
The teacher gave the lecture
From our PCFG, $P(NP \rightarrow DT \ NN) = 0.33$

$[0,2] = [0,1] \times [1,2] \times P(NP \rightarrow DT \ NN) = 1.0 \times 0.5 \times 0.33 = 0.165$
From our PCFG, \( P(NP \rightarrow DT \ NN) = 0.33 \)
\[
\begin{align*}
[0,2] &= [0,1] \times [1,2] \times P(NP \rightarrow DT \ NN) = 1.0 \times 0.5 \times 0.33 = 0.165
\end{align*}
\]
From our PCFG, \( P(\text{VP} \rightarrow \text{VB NP}) = 0.25 \)

\[
[2,5] = [2,3] \times [3,5] \times P(\text{VP} \rightarrow \text{VP NP}) = 1.0 \times 0.5 \times 0.165 = 0.0825
\]
From our PCFG, $P(VP \rightarrow VB \ NP) = 0.25$

$[2,5] = [2,3] \times [3,5] \times P(VP \rightarrow VP \ NP) = 1.0 \times 0.5 \times 0.165 = 0.0825$
Decoding works similarly to before, but this time with more argmax, for deciding which possible branch is the “best.”

See Chapter 14 of J&M for details.
Performance considerations:

CYK is $O(N^3 \times |G|)$, where:

$N$ is the number of tokens, and...

$|G|$ is the size of the grammar (number of rules).

$|G|$ is often very large!

Therefore, CYK is often not super-fast.
Performance considerations:

The “slow part” involves grammar *intersection*:

Finding rules from the grammar that can produce a given set of children.

Optimizing CYK often comes down to figuring out clever ways to represent a grammar s.t. this is fast.
You will implement CYK in MPs 6–7…

Keep an eye on the website for details.