2) PCFGs and CKY parsing

Paula Buttery

Dept of Computer Science & Technology, University of Cambridge
Reminder: languages can also be defined using automata

Recall that a language is regular if it is equal to the set of strings accepted by some deterministic finite-state automaton (DFA). A DFA is defined as $M = (Q, \Sigma, \Delta, s, F)$ where:

- $Q = \{q_0, q_1, q_2...\}$ is a finite set of states.
- $\Sigma$ is the alphabet: a finite set of transition symbols.
- $\Delta \subseteq Q \times \Sigma \times Q$ is a function $Q \times \Sigma \rightarrow Q$ which we write as $\delta$. Given $q \in Q$ and $i \in \Sigma$ then $\delta(q, i)$ returns a new state $q' \in Q$.
- $s$ is a starting state.
- $F$ is the set of all end states.
Reminder: **regular languages** are accepted by **DFAs**

For $L(M) = \{ a, ab, abb, \ldots \}$:

$M = (Q = \{ q_0, q_1, q_2 \},$

$\Sigma = \{ a, b \},$

$\Delta = \{ (q_0, a, q_1), (q_0, b, q_2), \ldots, (q_2, b, q_2) \},$

$s = q_0,$

$F = \{ q_1 \}$)
Regular grammars

Simple relationship between a DFA and production rules

\[ Q = \{ S, A, B, C, q_4 \} \]
\[ \Sigma = \{ b, a, ! \} \]
\[ q_0 = S \]
\[ F = \{ q_4 \} \]

\[ S \rightarrow bA \]
\[ A \rightarrow aB \]
\[ B \rightarrow aC \]
\[ C \rightarrow aC \]
\[ C \rightarrow ! \]
Regular grammars generate regular languages

Given a DFA $M = (Q, \Sigma, \Delta, s, F)$ the language, $\mathcal{L}(M)$, of strings accepted by $M$ can be generated by the regular grammar $G_{reg} = (N, \Sigma, S, P)$ where:

- $N = Q$ the non-terminals are the states of $M$
- $\Sigma = \Sigma$ the terminals are the set of transition symbols of $M$
- $S = s$ the starting symbol is the starting state of $M$
- $P = q_i \rightarrow aq_j$ when $\delta(q_i, a) = q_j \in \Delta$
  or $q_i \rightarrow \epsilon$ when $q \in F$ (i.e. when $q$ is an end state)
Strings are **derived** from production rules

In order to derive a string from a grammar
- start with the designated starting symbol
- then non-terminal symbols are repeatedly expanded using the rewrite rules until there is nothing further left to expand.

The rewrite rules derive the members of a language from their internal structure (or **phrase structure**)
A regular language has a **left-** and **right-linear** grammar

For every regular grammar the rewrite rules of the grammar can all be expressed in the form:

\[
X \rightarrow aY \\
X \rightarrow a
\]

or alternatively, they can all be expressed as:

\[
X \rightarrow Ya \\
X \rightarrow a
\]

The two grammars are **weakly-equivalent** since they generate the same strings.
But not **strongly-equivalent** because they do not generate the same structure to strings.
A regular language has a **left-** and **right-linear** grammar.

For every regular grammar the rewrite rules of the grammar can all be expressed in the form:

\[
\begin{align*}
X & \rightarrow aY \\
X & \rightarrow a
\end{align*}
\]

or alternatively, they can all be expressed as:

\[
\begin{align*}
X & \rightarrow Ya \\
X & \rightarrow a
\end{align*}
\]

The two grammars are **weakly-equivalent** since they generate the same strings.

But not **strongly-equivalent** because they do not generate the same structure to strings.
A regular language has a **left-** and **right-linear** grammar.

For every regular grammar the rewrite rules of the grammar can all be expressed in the form:

\[
\begin{align*}
X & \rightarrow aY \\
X & \rightarrow a \\
\end{align*}
\]

or alternatively, they can all be expressed as:

\[
\begin{align*}
X & \rightarrow Ya \\
X & \rightarrow a \\
\end{align*}
\]

The two grammars are **weakly-equivalent** since they generate the same strings.

But not **strongly-equivalent** because they do not generate the same structure to strings.
A regular language has a **left- and right-linear** grammar

Regular grammars

A regular language has a **left- and right-linear** grammar.

**Figure 6**: Structures for the smallest element of the language from the weakly equivalent left and right-linear grammars for sheep talk.

Using a regular grammar to model natural language

There are other forms of formal grammar that vary in the combination of terminal and non-terminals permitted either side of the production arrow. We shall come back to these in lecture 8 when we discuss the complexity of different grammars. Broadly speaking, the less restrictive you are about the form of the production rules the more expressive your language can be; but this usually has the side effect of increasing the search space and the complexity of the search algorithm (more on that later). For now the question is simply whether we need anything more expressive than the regular grammar we have discussed in order to model natural language.

The answer is that we probably do need something more expressive for several reasons:

**Centre Embedding** - The syntax of natural languages cannot be described by an FSA, even in principle, due to the presence of centre-embedding; i.e. infinitely recursive structures described by the rule, $A \rightarrow \alpha A \beta$, which generate language examples of the form, $a^n b^n$. For instance, Sentence 1 has a centre-embedded structure. J&M provide the example shown in Sentence 2.

See J&M section 12, page 447.

Sentence 1: The students the police arrested complained.

Sentence 2: The luggage that the passengers checked arrived.

Sentence 3: The luggage that the passengers that the storm delayed checked arrived.

The reason that an FSA cannot describe centre-embedding is that it has no memory of what has occurred previously in the sentence. In order to ‘know’ that $n$ verbs were required to match $n$ nominals already seen, the FSA would have to ‘record’ that $n$ nominals had been seen; but the FSA has no mechanism to do this. However, examples of centre-embedding quickly become unwieldy for human processing (n.b. the difficulty of understanding Sentence 3). For finite $n$ we can still model the language using an FSA: we can design the states to capture finite levels.
A regular grammar is a **phrase structure grammar**

A phrase structure grammar over an alphabet $\Sigma$ is defined by a tuple $G = (\mathcal{N}, \Sigma, S, \mathcal{P})$. The language generated by grammar $G$ is $\mathcal{L}(G)$:

**Non-terminals $\mathcal{N}$**: Non-terminal symbols (often uppercase letters) may be **rewritten** using the rules of the grammar.

**Terminals $\Sigma$**: Terminal symbols (often lowercase letters) are elements of $\Sigma$ and **cannot be rewritten**. Note $\mathcal{N} \cap \Sigma = \emptyset$.

**Start Symbol $S$**: A **distinguished non-terminal symbol** $S \in \mathcal{N}$. This non-terminal provides the starting point for derivations.

**Phrase Structure Rules $\mathcal{P}$**: Phrase structure rules are pairs of the form $(w, v)$ usually written:

$w \rightarrow v$, where $w \in (\Sigma \cup \mathcal{N})^* \mathcal{N} (\Sigma \cup \mathcal{N})^*$ and $v \in (\Sigma \cup \mathcal{N})^*$
Definition of a phrase structure grammar **derivation**

Given $G = (\mathcal{N}, \Sigma, S, \mathcal{P})$ and $w, v \in (\mathcal{N} \cup \Sigma)^*$ a **derivation step** is possible to transform $w$ into $v$ if:

$u_1, u_2 \in (\mathcal{N} \cup \Sigma)^*$ exist such that $w = u_1\alpha u_2$, and $v = u_1\beta u_2$ and $\alpha \rightarrow \beta \in \mathcal{P}$

This is written $w \xrightarrow{G} v$

A string in the language $\mathcal{L}(G)$ is a member of $\Sigma^*$ that can be derived in a **finite number of derivation steps** from the starting symbol $S$.

We use $\Rightarrow$ to denote the reflexive, transitive closure of derivation steps, consequently $\mathcal{L}(G) = \{w \in \Sigma^* | S \xrightarrow{G^*} w\}$. 
PSGs may be grouped by production rule properties

Chomsky suggested that phrase structure grammars may be grouped together by the properties of their production rules.

<table>
<thead>
<tr>
<th>Name</th>
<th>Form of Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>regular</td>
<td>$(A \rightarrow Aa \text{ or } A \rightarrow aA)$ and $A \rightarrow a \mid A \in \mathcal{N} \text{ and } a \in \Sigma$</td>
</tr>
<tr>
<td>context-free</td>
<td>$A \rightarrow \alpha \mid A \in \mathcal{N} \text{ and } \alpha \in (\mathcal{N} \cup \Sigma)^*$</td>
</tr>
<tr>
<td>context-sensitive</td>
<td>$\alpha A\beta \rightarrow \alpha \gamma \beta \mid A \in \mathcal{N} \text{ and } \alpha, \beta, \gamma \in (\mathcal{N} \cup \Sigma)^* \text{ and } \gamma \neq \epsilon$</td>
</tr>
<tr>
<td>recursively enum</td>
<td>$\alpha \rightarrow \beta \mid \alpha, \beta \in (\mathcal{N} \cup \Sigma)^* \text{ and } \alpha \neq \epsilon$</td>
</tr>
</tbody>
</table>

A class of languages (e.g. the class of regular languages) is all the languages that can be generated by a particular type of grammar.

The term power is used to describe the expressivity of each type of grammar in the hierarchy (measured in terms of the number of subsets of $\Sigma^*$ that the type can generate).
We can define the **complexity** of language classes.

The **complexity** of a language class is defined in terms of the **recognition** problem.

<table>
<thead>
<tr>
<th>Type</th>
<th>Language Class</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>regular</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>2</td>
<td>context-free</td>
<td>$O(n^c)$</td>
</tr>
<tr>
<td>1</td>
<td>context-sensitive</td>
<td>$O(c^n)$</td>
</tr>
<tr>
<td>0</td>
<td>recursively enumerable</td>
<td><strong>undecidable</strong></td>
</tr>
</tbody>
</table>
Context-free grammars capture *constituency*

$$G = (\mathcal{N}, \Sigma, S, \mathcal{P}) \text{ where } \mathcal{P} = \{ A \to \alpha \mid A \in \mathcal{N}, \alpha \in (\mathcal{N} \cup \Sigma)^* \}$$
CFGs can be written in **Chomsky Normal Form**

**Chomsky normal form**: every production rule has the form, \( A \to BC \), or, \( A \to a \) where \( A, B, C \in \mathcal{N} \), and, \( a \in \Sigma \).

**Conversion to Chomsky Normal Form**

For every CFG there is a weakly equivalent CNF alternative. \( A \to BCD \) may be rewritten as the two rules, \( A \to BX \), and, \( X \to CD \).
CFGs can be written in **Chomsky Normal Form**

For \( A, B, C, D, X, Y \in \mathcal{N} \) and \( \gamma, \beta \subseteq \mathcal{N}^* \) and \( a \in \Sigma \).

**Conversion to Chomsky Normal Form**

- Keep all existing conforming rules
- replace \( A \rightarrow \gamma a \beta \) with \( D \rightarrow \gamma A \beta \) and \( A \rightarrow a \)
- repeatedly replace \( A \rightarrow \gamma BC \) with \( A \rightarrow \gamma X \) and \( X \rightarrow BC \)
- if \( A \Rightarrow B \) is a chain of one or more unit productions, and \( B \rightarrow a \) then replace all the unit productions with \( A \rightarrow a \) (where a unit production is any rule of the form \( X \rightarrow Y \))

CNF is a requirement for the CKY parsing algorithm but it causes some problems:

- Grammar is no longer linguistically intuitive
- Direct correspondence with compositional semantics may be lost
CFGs can be written in **Chomsky Normal Form**

For $A, B, C, D, X, Y \in \mathcal{N}$ and $\gamma, \beta \subseteq \mathcal{N}^*$ and $a \in \Sigma$.

**Conversion to Chomsky Normal Form**

- Keep all existing conforming rules
- replace $A \rightarrow \gamma a \beta$ with $D \rightarrow \gamma A \beta$ and $A \rightarrow a$
- repeatedly replace $A \rightarrow \gamma BC$ with $A \rightarrow \gamma X$ and $X \rightarrow BC$
- if $A \Rightarrow B$ is a chain of one or more unit productions, and $B \rightarrow a$ then replace all the unit productions with $A \rightarrow a$ (where a unit production is any rule of the form $X \rightarrow Y$)

CNF is a requirement for the CKY parsing algorithm but it causes some problems:

- Grammar is no longer linguistically intuitive
- Direct correspondence with compositional semantics may be lost
Deterministic context-free languages:
- are a proper subset of the context-free languages
- can be modelled by an unambiguous grammar
- can be parsed in linear time
- parser can be automatically generated from the grammar

Natural languages (with all their inherent ambiguity) are not well suited to algorithms which operate deterministically recognising a single derivation without backtracking.

However, natural language parsing can be achieved deterministically by selecting parsing actions using a machine learning classifier (more on this in later lectures).

All CFLs (including those exhibiting ambiguity) can be recognised in polynomial time using dynamic programming algorithms.
CFGs used to model natural language are **not** deterministic

**Deterministic** context-free languages:

- are a proper subset of the context-free languages
- can be modelled by an unambiguous grammar
- can be parsed in linear time
- parser can be automatically generated from the grammar

CFGs used to model natural language are **not** deterministic

- Natural languages (with all their inherent ambiguity) are not well suited to algorithms which operate deterministically recognising a single derivation without backtracking

  - However, natural language parsing can be achieved deterministically by selecting parsing actions using a machine learning classifier (more on this in later lectures).
  - All CFLs (including those exhibiting ambiguity) can be recognised in polynomial time using **dynamic programming algorithms**.
CFGs used to model natural language are not deterministic

Deterministic context-free languages:
- are a proper subset of the context-free languages
- can be modelled by an unambiguous grammar
- can be parsed in linear time
- parser can be automatically generated from the grammar

Natural languages (with all their inherent ambiguity) are not well suited to algorithms which operate deterministically recognising a single derivation without backtracking

However, natural language parsing can be achieved deterministically by selecting parsing actions using a machine learning classifier (more on this in later lectures).

All CFLs (including those exhibiting ambiguity) can be recognised in polynomial time using dynamic programming algorithms.
CFGs used to model natural language are \textbf{not} deterministic

**Deterministic** context-free languages:
- are a proper subset of the context-free languages
- can be modelled by an unambiguous grammar
- can be parsed in linear time
- parser can be automatically generated from the grammar

CFGs used to model natural language are \textbf{not} deterministic
- Natural languages (with all their inherent ambiguity) are not well suited to algorithms which operate deterministically recognising a single derivation without backtracking
- However, natural language parsing can be achieved deterministically by selecting parsing actions using a machine learning classifier (more on this in later lectures).
- All CFLs (including those exhibiting ambiguity) can be recognised in polynomial time using \textit{dynamic programming algorithms}. 
The CKY algorithm *recognises* strings in a CFL

<table>
<thead>
<tr>
<th></th>
<th>they</th>
<th>1</th>
<th>can</th>
<th>2</th>
<th>fish</th>
<th>3</th>
</tr>
</thead>
</table>

**Toy CNF grammar:**

\[
\begin{align*}
N & = \{ S, NP, VP, VV, VM \} \\
\Sigma & = \{ can, fish, they \} \\
S & = S \\
\mathcal{P} & = \{ S \to NP \ VP \\
& \quad VP \to VM \ VV \\
& \quad VP \to VV \ NP \\
& \quad VV \to can \mid fish \\
& \quad VM \to can \\
& \quad NP \to they \mid fish \} \\
\end{align*}
\]

String is in the language when the cell \([0, 3]\) contains \(S\)
The CKY algorithm recognizes strings in a CFL.

They can fish.

Toy CNF grammar:

\[
\mathcal{N} = \{ S, NP, VP, VV, VM \} \\
\Sigma = \{ can, fish, they \} \\
S = S \\
\mathcal{P} = \{ S \rightarrow NP \ VP \\
VP \rightarrow VM \ VV \\
VP \rightarrow VV \ NP \\
VV \rightarrow can \mid fish \\
VM \rightarrow can \\
NP \rightarrow they \mid fish \} 
\]

String is in the language when the cell \([0, 3]\) contains \(S\).
The CKY algorithm recognises strings in a CFL

Toy CNF grammar:

\[ \mathcal{N} = \{ S, NP, VP, VV, VM \} \]
\[ \Sigma = \{ can, fish, they \} \]
\[ S = S \]
\[ \mathcal{P} = \{ S \to NP \ VP \\ VP \to VM \ VV \\ VP \to VV \ NP \\ VV \to can \ | \ fish \\ VM \to can \\ NP \to they \ | \ fish \} \]

String is in the language when the cell \([0, 3]\) contains \(S\)
The CKY algorithm *recognises* strings in a CFL

```
0  they  1  can  2  fish  3

TO
1  2  3

0  NP

FROM 1

2

VV

VM

String is in the language when the cell \([0, 3]\) contains \(S\)
```

Toy CNF grammar:

\[
\begin{align*}
\mathcal{N} &= \{ S, NP, VP, VV, VM \} \\
\Sigma &= \{ can, fish, they \} \\
S &= S \\
\mathcal{P} &= \{ S \rightarrow NP\ VP \\
&\quad VP \rightarrow VM\ VV \\
&\quad VP \rightarrow VV\ NP \\
&\quad VV \rightarrow can \mid fish \\
&\quad VM \rightarrow can \\
&\quad NP \rightarrow they \mid fish \} 
\end{align*}
\]
The CKY algorithm **recognises** strings in a CFL

<table>
<thead>
<tr>
<th>0</th>
<th>they</th>
<th>1</th>
<th>can</th>
<th>2</th>
<th>fish</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FROM</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>.npy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Toy CNF grammar:

\[
\mathcal{N} = \{S, NP, VP, VV, VM\} \\
\Sigma = \{can, fish, they\} \\
S = S \\
\mathcal{P} = \{S \rightarrow NP \ VP \ \\
\quad VP \rightarrow VM \ VV \ \\
\quad VP \rightarrow VV \ NP \ \\
\quad VV \rightarrow can \mid fish \ \\
\quad VM \rightarrow can \ \\
\quad NP \rightarrow they \mid fish\} \]

String is in the language when the cell \([0, 3]\) contains \(S\)
The CKY algorithm recognizes strings in a CFL

Toy CNF grammar:

\[ \mathcal{N} = \{ S, NP, VP, VV, VM \} \]
\[ \Sigma = \{ can, fish, they \} \]
\[ S = S \]
\[ \mathcal{P} = \{ S \rightarrow NP \ VP \\
VP \rightarrow VM \ VV \\
VP \rightarrow VV \ NP \\
VV \rightarrow can \mid fish \\
VM \rightarrow can \\
NP \rightarrow they \mid fish \} \]

String is in the language when the cell \([0,3]\) contains \(S\)
The CKY algorithm recognizes strings in a CFL

Toy CNF grammar:

\[ N = \{ S, NP, VP, VV, VM \} \]
\[ \Sigma = \{ \text{can, fish, they} \} \]
\[ S = S \]
\[ \mathcal{P} = \{ S \rightarrow NP \ VP \}
\]
\[ VP \rightarrow VM \ VV \]
\[ VP \rightarrow VV \ NP \]
\[ VV \rightarrow \text{can} | \text{fish} \]
\[ VM \rightarrow \text{can} \]
\[ NP \rightarrow \text{they} | \text{fish} \} \]

String is in the language when the cell \([0,3]\) contains \(S\)
The CKY algorithm **recognises** strings in a CFL

<table>
<thead>
<tr>
<th>0</th>
<th>they</th>
<th>1</th>
<th>can</th>
<th>2</th>
<th>fish</th>
<th>3</th>
</tr>
</thead>
</table>

**Toy CNF grammar:**

\[
N = \{ S, NP, VP, VV, VM \} \\
\Sigma = \{ can, fish, they \} \\
S = S \\
P = \{ S \rightarrow NP \ VP \\
    VP \rightarrow VM\ VV \\
    VP \rightarrow VV\ NP \\
    VV \rightarrow can | fish \\
    VM \rightarrow can \\
    NP \rightarrow they | fish \} \\
\]

String is in the language when the cell \([0, 3]\) contains \(S\)
The CKY algorithm **recognises** strings in a CFL.

<table>
<thead>
<tr>
<th>0</th>
<th>they</th>
<th>1</th>
<th>can</th>
<th>2</th>
<th>fish</th>
<th>3</th>
</tr>
</thead>
</table>

**Toy CNF grammar:**

\[
\begin{align*}
\mathcal{N} &= \{ S, \ NP, \ VP, \ VV, \ VM \} \\
\Sigma &= \{ can, \ fish, \ they \} \\
S &= S \\
\mathcal{P} &= \{ S \rightarrow NP \ VP \\
VP &\rightarrow VM \ VV \\
VP &\rightarrow VV \ NP \\
VV &\rightarrow can \ | \ fish \\
VM &\rightarrow can \\
NP &\rightarrow they \ | \ fish \} \\
\end{align*}
\]

String is in the language when the cell \([0, 3]\) contains \(S\).
The CKY algorithm **recognises** strings in a CFL

In the general case for $A, B, C \in \mathcal{N}$ and $a \in \Sigma$:

- If $a \in \Sigma$ exists between indexes $m$ and $m+1$, and $A \rightarrow a$ then cell $[m, m+1]$ contains $A$
- If cell $[i, k]$ contains $B$ and cell $[k, j]$ contains $C$ and $A \rightarrow BC$ then cell $[i, j]$ contains $A$
- String of length $n$ is in the language when the cell $[0, n]$ contains $S$

The CKY algorithm only recognises a string, in order to obtain the parse tree we need to:

- pair each non-terminal in a cell with a 2-tuple of the cells that derived it
- allow the same non-terminal to exist more than once in any particular cell (or allow it to be paired with a list of 2-tuples)
The CKY algorithm **recognises** strings in a CFL

In the general case for $A, B, C \in \mathcal{N}$ and $a \in \Sigma$:

- If $a \in \Sigma$ exists between indexes $m$ and $m + 1$, and $A \rightarrow a$ then cell $[m, m + 1]$ contains $A$
- if cell $[i, k]$ contains $B$ and cell $[k, j]$ contains $C$ and $A \rightarrow BC$ then cell $[i, j]$ contains $A$
- String of length $n$ is in the language when the cell $[0, n]$ contains $S$

The CKY algorithm only recognises a string, in order to obtain the **parse tree** we need to:

- pair each non-terminal in a cell with a 2-tuple of the cells that derived it
- allow the same non-terminal to exist more than once in any particular cell (or allow it to be paired with a list of 2-tuples)
The CKY algorithm can be used to create a parse

```
  TO
  1   2   3

  FROM 1

  2

  they can fish
```
The CKY algorithm can be used to create a parse

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NP_{they}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FROM</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```

```
TO

1       2       3

they     can     fish
```
The CKY algorithm can be used to create a parse.

**Syntax Example:**

```
FROM 1

0  NP_{they}

1  VV_{can}

2

TO

1  2  3

VV_{can}

they  can  fish
```

**Statement:**

```
1

S \rightarrow ([0, 1] NP, [1, 3] VP)

S \rightarrow ([0, 1] NP, [1, 3] VP)
```

**Example Sentence:**

```
can (can) (can)

NP (they)

FROM 1

VV (can)

VM (can)
```

**Sentence:**

```
they can fish
```
The CKY algorithm can be used to create a parse tree for the sentence "they can fish."
The CKY algorithm can be used to create a parse

\[ T_0 \rightarrow NP_{(they)} \]

\[ F_1 \rightarrow VV_{(can)} \]
\[ VV_{(can)} \rightarrow VM_{(can)} \]

\[ F_2 \rightarrow VV_{(fish)} \]
\[ VV_{(fish)} \rightarrow NP_{(fish)} \]

they can fish
The CKY algorithm can be used to create a parse

```
TO

1  2  3

0  NP_{(they)}

FROM  1

1  VV_{(can)}  VP_1 \rightarrow ([1,2]VV,[2,3]NP)

2  VM_{(can)}  VP_2 \rightarrow ([1,2]VM,[2,3]VV)

2  VV_{(fish)}

NP_{(fish)}

they  can  fish
```
The CKY algorithm can be used to create a parse.

```
               TO
              1   2   3

0  NP_{they} .  S_1 \rightarrow ([0,1]_{NP}, [1,3]_{VP_1})

FROM  1
       2

1  VV_{can}  VP_1 \rightarrow ([1,2]_{VV}, [2,3]_{NP})

VM_{can}  VP_2 \rightarrow ([1,2]_{VM}, [2,3]_{VV})

2  NP_{fish}  VV_{fish}

they  can  fish
```
Ambiguous grammars derive a **parse forest**

Number of binary trees is proportional to the Catalan number

\[
\text{Num of trees for sentence length } n = \prod_{k=2}^{n-1} \frac{(n - 1) + k}{k}
\]

<table>
<thead>
<tr>
<th>sentence length</th>
<th>number of trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>132</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sentence length</th>
<th>number of trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>429</td>
</tr>
<tr>
<td>9</td>
<td>1430</td>
</tr>
<tr>
<td>10</td>
<td>4862</td>
</tr>
<tr>
<td>11</td>
<td>16796</td>
</tr>
<tr>
<td>12</td>
<td>58786</td>
</tr>
</tbody>
</table>

We need parsing algorithms that can efficiently store the parse forest and not derive shared parts of tree more than once—
Ambiguous grammars derive a *parse forest*

Number of binary trees is proportional to the Catalan number

$$\text{Num of trees for sentence length } n = \prod_{k=2}^{n-1} \frac{(n-1)+k}{k}$$

<table>
<thead>
<tr>
<th>sentence length</th>
<th>number of trees</th>
<th>sentence length</th>
<th>number of trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>8</td>
<td>429</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>9</td>
<td>1430</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>10</td>
<td>4862</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>11</td>
<td>16796</td>
</tr>
<tr>
<td>7</td>
<td>132</td>
<td>12</td>
<td>58786</td>
</tr>
</tbody>
</table>

We need parsing algorithms that can efficiently store the parse forest and not derive shared parts of tree more than once—
Ambiguous grammars derive a parse forest

Number of binary trees is proportional to the Catalan number

\[
\text{Num of trees for sentence length } n = \prod_{k=2}^{n-1} \frac{(n - 1) + k}{k}
\]

<table>
<thead>
<tr>
<th>sentence length</th>
<th>number of trees</th>
<th>sentence length</th>
<th>number of trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>8</td>
<td>429</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>9</td>
<td>1430</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>10</td>
<td>4862</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
<td>11</td>
<td>16796</td>
</tr>
<tr>
<td>7</td>
<td>132</td>
<td>12</td>
<td>58786</td>
</tr>
</tbody>
</table>

We need parsing algorithms that can efficiently store the parse forest and not derive shared parts of tree more than once—use packing and/or a beam (the latter requires knowledge of the probability of derivations)
Parse probabilities may be derived using a **PCFG**

- \( G_{pcfg} = (\Sigma, \mathcal{N}, S, \mathcal{P}, q) \) where \( q \) is a mapping from rules in \( \mathcal{P} \) to a probability and \( \sum_{A \rightarrow \alpha \in \mathcal{P}} q(A \rightarrow \alpha) = 1 \)

- \( G_{pcfg} \) is **consistent** if the sum of all probabilities of all derivable strings equals 1 (grammars with infinite loops like \( S \rightarrow S \) are inconsistent)

- The probability of a particular parse is the **product** of the probabilities of the rules that defined the parse tree. For a string \( W \) with parse tree \( T \) derived from rules \( A_i \rightarrow B_i, \ i = 1...n \)

  \[
P(T, W) = \prod_{i=1}^{n} P(A_i \rightarrow B_i)
  \]

- But note that \( P(T, W) = P(T)P(W|T) \) and that \( P(W|T) = 1 \) so

  \[
P(T, W) = P(T) \text{ and thus } P(T) = \prod_{i=1}^{n} P(A_i \rightarrow B_i)
  \]
Parse probabilities may be derived using a **PCFG**

- The probability of an ambiguous string is the sum of all the parse trees that **yield** that string.

\[
P(W) = \sum_{\text{trees that yield } W} P(T, W) = \sum_{\text{trees that yield } W} P(T)
\]

- We can disambiguate multiple parses by choosing the most probable parse tree for the string:

\[
\hat{T}(W) = \arg\max_{\text{trees that yield } W} P(T|W)
\]

but

\[
P(T|W) = \frac{P(T, W)}{P(W)} \rightarrow P(T, W) = P(T)
\]

so

\[
\hat{T}(W) = \arg\max_{\text{trees that yield } W} P(T)
\]
Rule probabilities may be estimated from treebanks

- A **treebank** is a corpus of parsed sentences
- Rule probabilities can be estimated from counts in a treebank:
  \[ P(A \rightarrow B) = \frac{\text{count}(A \rightarrow B)}{\sum_{\gamma} \text{count}(A \rightarrow \gamma)} = \frac{\text{count}(A \rightarrow B)}{\text{count}(A)} \]
- **inside-outside algorithm** can be used when no tree bank exists...

Problems with PCFGs:
- Independence ignores structural dependency within the tree
- Structure is dependent on lexical items...
A **treebank** is a corpus of parsed sentences.

Rule probabilities can be estimated from counts in a treebank:

\[ P(A \rightarrow B) = P(A \rightarrow B | A) = \frac{\text{count}(A \rightarrow B)}{\sum_{\gamma} \text{count}(A \rightarrow \gamma)} = \frac{\text{count}(A \rightarrow B)}{\text{count}(A)} \]

**inside-outside algorithm** can be used when no tree bank exists.

... more in later lectures.

Problems with PCFGs:

- Independence ignores structural dependency within the tree.
- Structure is dependent on lexical items.

... more in later lectures.
Rule probabilities may be estimated from treebanks

- A **treebank** is a corpus of parsed sentences
- Rule probabilities can be estimated from counts in a treebank:
  \[ P(A \rightarrow B) = P(A \rightarrow B | A) = \frac{\text{count}(A \rightarrow B)}{\sum_{\gamma} \text{count}(A \rightarrow \gamma)} = \frac{\text{count}(A \rightarrow B)}{\text{count}(A)} \]

- **inside-outside algorithm** can be used when no tree bank exists

Problems with PCFGs:
- Independence ignores structural dependency within the tree
- Structure is dependent on lexical items
Rule probabilities may be estimated from treebanks

- A treebank is a corpus of parsed sentences
- Rule probabilities can be estimated from counts in a treebank:
  \[ P(A \rightarrow B) = P(A \rightarrow B | A) = \frac{\text{count}(A \rightarrow B)}{\sum_{\gamma} \text{count}(A \rightarrow \gamma)} = \frac{\text{count}(A \rightarrow B)}{\text{count}(A)} \]
- inside-outside algorithm can be used when no tree bank exists

Problems with PCFGs:
- Independence ignores structural dependency within the tree
- Structure is dependent on lexical items
Probabilistic CFGs may be incorporated into CKY

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ N = \{ S, NP, VP, VV, VM \} \]
\[ \Sigma = \{ can, fish, they \} \]
\[ S = S \]
\[ P = \{ S \rightarrow NP \ VP \ 1.0 \]
\[ VP \rightarrow VM \ VV \ 0.9 \]
\[ VP \rightarrow VV \ NP \ 0.1 \]
\[ VV \rightarrow can \ 0.2 \ | \ fish \ 0.8 \]
\[ VM \rightarrow can \ 1.0 \]
\[ NP \rightarrow they \ 0.5 \ | \ fish \ 0.5 \]

- For the best parse keep most probable non-terminal at each node
- Otherwise can pack and operate a beam
Probabilistic CFGs may be incorporated into CKY

For the best parse keep most probable non-terminal at each node

Otherwise can pack and operate a beam
Probabilistic CFGs may be incorporated into CKY

For the best parse keep most probable non-terminal at each node

Otherwise can pack and operate a beam
Probabilistic CFGs may be incorporated into CKY

For the best parse keep most probable non-terminal at each node

Otherwise can pack and operate a beam
Probabilistic CFGs may be incorporated into CKY

For the best parse keep most probable non-terminal at each node

- Otherwise can pack and operate a beam
Probabilistic CFGs may be incorporated into CKY

For the best parse keep most probable non-terminal at each node
Otherwise can pack and operate a beam
Probabilistic CFGs may be incorporated into CKY

For the best parse keep most probable non-terminal at each node
Otherwise can pack and operate a beam