Lecture 18: Expressive Grammars

Julia Hockenmaier
juliahmr@illinois.edu
3324 Siebel Center
Lecture 18:
Expressive Grammars

Part 1: Grammars in NLP: what and why
What is grammar?

Grammar formalisms
(= linguists’ programming languages)
A precise way to define and describe the structure of sentences.
(N.B.: There are many different formalisms out there, which each define their own data structures and operations)

Specific grammars
(= linguists’ programs)
Implementations (in a particular formalism) for a particular language (English, Chinese,....)
(NB: any practical parser will need to also have a model/scoring function to identify which grammatical analysis should be assigned to a given sentence)
Why study grammar?

Linguistic questions:
What kind of constructions occur in natural language(s)?

Formal questions:
Can we define formalisms that allow us to characterize which strings belong to a language?
Those formalisms have appropriate weak generative capacity
Can we define formalisms that allow us to map sentences to their appropriate structures?
Those formalisms have appropriate strong generative capacity

Practical applications (Syntactic/Semantic Parsing):
Can we identify the grammatical structure of sentences?
Can we translate sentences to appropriate meaning representations?
Can we define a program that generates all English sentences?

Undergeneration

John saw Mary.
I ate sushi with tuna.

I want you to go there.
I ate the cake that John had made for me yesterday

Did you go there?

John made some cake.

Did you went there?

Overgeneration

English

John Mary saw.

with tuna sushi ate I.

Did you went there?
Syntax as an interface to semantics

Surface string: Mary saw John

Grammar

Parsing

Logical form: saw(Mary, John)

Pred-arg structure:

- PRED: saw
- AGENT: Mary
- PATIENT: John

Dependency graph:

saw

Mary -- John
Grammar formalisms

Formalisms provide a formal language in which linguistic theories can be expressed and implemented.

Formalisms define elementary objects (trees, strings, feature structures) and recursive operations which generate complex objects from simple objects.

Different formalisms may impose different constraints (e.g. on the kinds of dependencies they can capture).
What makes a formalism “expressive”?

“Expressive” formalisms are richer than context-free grammars.

Different formalisms use different mechanisms, data structures and operations to go beyond CFGs
Examples of expressive grammar formalisms

**Tree-adjoining Grammar (TAG):**
Fragments of phrase-structure trees

**Combinatory Categorial Grammar (CCG):**
Syntactic categories paired with meaning representations

**Lexical-functional Grammar (LFG):**
Annotated phrase-structure trees (c-structure) linked to feature structures (f-structure)

**Head-Driven Phrase Structure Grammar (HPSG):**
Complex feature structures (Attribute-value matrices)
Lecture 18:
Expressive Grammars

Part 2:
Why go beyond CFGs?
The dependencies so far:

Arguments:
Verbs take arguments: subject, object, complements, ...
**Heads subcategorize for their arguments**

Adjuncts/Modifiers:
Adjectives modify nouns, adverbs modify VPs or adjectives, PPs modify NPs or VPs
**Modifiers subcategorize for the head**

Typically, these are *local* dependencies: they can be expressed *within individual CFG rules*

\[ \text{VP} \rightarrow \text{Adv} \text{ Verb} \text{ NP} \]
Context-free grammars

CFGs capture only **nested** dependencies
- The dependency graph is a **tree**
- The dependencies **do not cross**
German: center embedding

...daß ich [Hans schwimmen] sah
...that I Hans swim saw
...that I saw [Hans swim]

...daß ich [Maria [Hans schwimmen] helfen] sah
...that I Maria Hans swim help saw
...that I saw [Mary help [Hans swim]]

...daß ich [Anna [Maria [Hans schwimmen] helfen] lassen] sah
...that I Anna Maria Hans swim help let saw
...that I saw [Anna let [Mary help [Hans swim]]]
Dependency structures in general

Nested (projective) dependency trees (CFGs)

Non-projective dependency trees

Non-local dependency graphs
Beyond CFGs: Nonprojective dependencies

Dependencies form a tree with crossing branches.
Dutch: Cross-Serial Dependencies

...dat ik Hans zag zwemmen
...that I Hans saw swim
...that I saw [Hans swim]

...dat ik Maria Hans zag helpen zwemmen
...that I Maria Hans saw help swim
...that I saw [Mary help [Hans swim]]

...dat ik Anna Maria Hans zag laten helpen zwemmen
...that I Anna Maria Hans saw let help swim
...that I saw [Anna let [Mary help [Hans swim]]]

Such cross-serial dependencies require mildly context-sensitive grammars
Other crossing (non-projective) dependencies

(Non-local) scrambling: In a sentence with multiple verbs, the argument of a verb appears in a different clause from that which contains the verb (arises in languages with freer word order than English)

*Die Pizza* hat Klaus versprochen zu *bringen*

The pizza has Klaus promised to bring

*Klaus has promised to bring the pizza*

Extraposition: Here, a modifier of the subject NP is moved to the end of the sentence

*The guy is coming* who is wearing a hat

Compare with the non-extraposed variant

*The [guy [who is wearing a hat]] is coming*

Topicalization: Here, the argument of the embedded verb is moved to the front of the sentence.

*Cheeseburgers, I [thought [he likes]]*
Beyond CFGs:
Nonlocal dependencies

Dependencies form a **DAG**
(a node may have **multiple incoming edges**)

Arise in the following constructions:
- **Control** *(He has promised me to go)*, **raising** *(He seems to go)*
- **Wh-movement** *(the man who you saw yesterday is here again)*,
- **Non-constituent** coordination
  (right-node raising, gapping, argument-cluster coordination)
Wh-Extraction (e.g. in English)

Relative clauses:
the sushi that [you told me [John saw [Mary eat]]]

Wh-Questions:
‘what [did you tell me [John saw [Mary eat]]]?’

Wh-questions (what, who, …) and relative clauses contain so-called unbounded nonlocal dependencies because the verb that subcategorizes for the moved NP may be arbitrarily deeply embedded in the tree. Linguists call this phenomenon wh-extraction (wh-movement).
As a phrase structure tree:
The trace analysis of wh-extraction
Slash categories for wh-extraction

Because only one element can be extracted, we can use **slash categories**.

This is still a CFG: the set of nonterminals is finite.

---

Generalized Phrase Structure Grammar (GPSG), Gazdar et al. (1985)
Lecture 18:
Expressive Grammars

Part 3:
Tree-Adjoining Grammar
(Lexicalized) Tree-Adjoining Grammar

TAG is a tree-rewriting formalism:

- TAG defines operations (substitution, adjunction) on trees.
- The elementary objects in TAG are trees (not strings)

TAG is lexicalized:

- Each elementary tree is anchored to a lexical item (word)
- “Extended domain of locality”:
  - The elementary tree contains all arguments of the anchor.
  - TAG requires a linguistic theory which specifies the shape of these elementary trees.

TAG is mildly context-sensitive:

- can capture Dutch cross-serial dependencies
- but is still efficiently parseable
Mildly context-sensitive grammars

Contain all context-free grammars/languages

Can be parsed in polynomial time (TAG/CCG: $O(n^6)$)

(Strong generative capacity) capture certain kinds of dependencies: nested (like CFGs) and cross-serial (like the Dutch example), but not the MIX language:
MIX: the set of strings $w \in \{a, b, c\}^*$ that contain equal numbers of as, bs and cs

Have the constant growth property:
the length of strings grows in a linear way
The power-of-2 language $\{a^{2n}\}$ does not have the constant growth property.
The Chomsky Hierarchy

- Regular
- Context-free
- Mildly context-sensitive
- Context-sensitive
- Recursively enumerable
Extended domain of locality

We want to capture all arguments of a word in a single elementary object.

We also want to retain certain syntactic structures (e.g. VPs).

Our elementary objects are tree fragments:
TAG substitution (arguments)

\[ \alpha_1: X \downarrow Y \]

\[ \alpha_2: X \]

\[ \alpha_3: Y \]

\[ \alpha_2 \]

\[ \alpha_3 \]

\[ \alpha_1 \]

\[ \alpha_2 \]

\[ \alpha_3 \]

\[ X \]

\[ Y \]

\[ \text{Derived tree:} \]

\[ \text{Substitute} \]

\[ \text{Derivation tree:} \]
TAG adjunction

Auxiliary tree

Derived tree:

Foot node

ADJOIN

Derivation tree:
The effect of adjunction

No adjunction: TSG (Tree substitution grammar)
TSG is context-free

Sister adjunction: TIG (Tree insertion grammar)
TIG is also context-free, but has a linguistically more adequate treatment of modifiers

Wrapping adjunction: TAG (Tree-adjoining grammar)
TAG is mildly context-sensitive
A small TAG lexicon

\[ \alpha_2: \]
\[ \text{NP} \]
\[ \text{John} \]

\[ \alpha_3: \]
\[ \text{NP} \]
\[ \text{tapas} \]

\[ \alpha_1: \]
\[ \text{S} \]
\[ \text{NP} \]
\[ \text{VP} \]
\[ \text{VBZ} \]
\[ \text{NP} \]
\[ \text{eats} \]

\[ \beta_1: \]
\[ \text{RB} \]
\[ \text{VP} \]
\[ \text{VP}^* \]
\[ \text{always} \]
A TAG derivation

\[ S \rightarrow NP \rightarrow VP \rightarrow VBZ \rightarrow \text{eats} \]

\[ \alpha_1: \]

\[ \beta_1: \]

\[ \alpha_2: \]

\[ \alpha_3: \]

\[ \text{always} \]

\[ \text{tapas} \]
A TAG derivation

\[
S \rightarrow NP \quad VP
\]

\[
NP \rightarrow John \quad NP \quad eats \quad NP \quad tapas
\]

\[
VP \rightarrow RB \quad VP^* \quad \beta_1
\]

\[
\alpha_1 \quad \alpha_2 \quad \beta_1 \quad \alpha_3
\]
A TAG derivation

\[
S \rightarrow \text{NP} \quad \text{VP} \quad \text{VP}^*
\]

\[
\text{NP} \quad \text{RB} \quad \text{VP} \\
\text{John} \quad \text{always} \quad \text{VBZ} \quad \text{NP}
\]

\[
\text{NP} \\
\text{eats} \quad \text{tapas}
\]
\(a^n b^n\): Cross-serial dependencies

Elementary trees:

Deriving \(aabb\)
Part 4:
(Combinatory)
Categorial Grammar
CCG: the machinery

Categories:
   specify subcat lists of words/constituents.

Combinatory rules:
   specify how constituents can combine.

The lexicon:
   specifies which categories a word can have.

Derivations:
   spell out process of combining constituents.
## CCG categories

### Simple (atomic) categories: NP, S, PP

### Complex categories (functions):

- **Return a result** when combined with an **argument**

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP, intransitive verb</td>
<td>$S \backslash NP$</td>
</tr>
<tr>
<td>Transitive verb</td>
<td>$(S \backslash NP) \backslash NP$</td>
</tr>
<tr>
<td>Adverb</td>
<td>$(S \backslash NP) \backslash (S \backslash NP)$</td>
</tr>
<tr>
<td>Prepositions</td>
<td>$((S \backslash NP) \backslash (S \backslash NP)) \backslash NP$</td>
</tr>
</tbody>
</table>
CCG categories are functions

CCG has a few atomic categories, e.g.

S, NP, PP

All other CCG categories are functions:

S / NP

Result Dir. Argument
Rules: Function application

\[
\frac{y \cdot y = x}{x}
\]
Rules: Function application

\[ \text{Result} \]

\[ S \]

\[ \text{Argument} \]
\[ \text{Function} \]

\[ y \cdot \frac{x}{y} = x \]
Rules: Function application

\[
\begin{array}{c}
\text{Result} \\
S\backslash NP \\
(S\backslash NP)/NP \\
\text{Function} \\
\text{Argument}
\end{array}
\]

\[
x \cdot y = x
\]
Function application

Forward application (>):

\[(S\backslash NP)/NP\quad NP\quad \Rightarrow >\quad S\backslash NP\]

\[eats\quad tapas\quad \Rightarrow >\quad eats\quad tapas\]

Backward application (<):

\[NP\quad S\backslash NP\quad \Rightarrow <\quad S\]

\[John\quad eats\quad tapas\quad \Rightarrow <\quad John\quad eats\quad tapas\]

Combines function \(X/Y\) or \(X\backslash Y\) with argument \(Y\) to yield result \(X\)
Used in all variants of categorial grammar
A (C)CG derivation

\[
\begin{array}{c}
\text{John} \\
\text{NP} \\
\hline \\
(S \backslash \text{NP})/\text{NP} \\
\hline \\
S \backslash \text{NP} \\
\hline \\
S \\
\end{array}
\]
Rules: Function Composition

\[
x \cdot y = x/z
\]

1\textsuperscript{st} Function

2\textsuperscript{nd} Function
Rules: Type-Raising

\[
S \rightarrow (S \backslash NP) \\
NP \\
\]

\[
y = \frac{x}{x} \cdot y = \frac{x}{\left(\frac{x}{y}\right)}
\]
Type-raising and composition

Type-raising:  \( X \rightarrow T/(T\backslash X) \)

Turns an argument into a function.

- NP  \( \rightarrow \) S/(S\backslash NP) (subject)
- NP  \( \rightarrow \) (S\backslash NP)/(S\backslash NP)/NP (object)

Harmonic composition:  \( X/Y \ Y/Z \rightarrow X/Z \)

Composes two functions (complex categories), same slashes
- (S\backslash NP)/PP  PP/NP  \( \rightarrow \) (S\backslash NP)/NP
- S/(S\backslash NP)  (S\backslash NP)/NP  \( \rightarrow \) S/NP

Crossing composition:  \( X/Y \ Y\backslash Z \rightarrow X\backslash Z \)

Composes two functions (complex categories), different slashes
- (S\backslash NP)/S  S\backslash NP  \( \rightarrow \) (S\backslash NP)\backslash NP
Type-raising and composition

Wh-movement (relative clause):

Right-node raising:
Lecture 18:
Expressive Grammars

Part 5:
Feature Structure Grammars
Why feature structures

Feature structures form the basis for many grammar formalisms used in computational linguistics.

Feature structure grammars (aka attribute-value grammars, or unification grammars) can be used as
- a more compact way of representing rich CFGs
- a way to represent more expressive grammars
Simple grammars overgenerate

\[
S \rightarrow NP \ VP \\
VP \rightarrow Verb \ NP \\
NP \rightarrow Det \ Noun \\
Det \rightarrow the | a | these \\
Verb \rightarrow eat | eats \\
Noun \rightarrow cake | cakes | student | students
\]

This generates ungrammatical sentences like

“these student eats a cakes”

We need to capture (number/person) agreement
Refining the nonterminals

\[
\begin{align*}
S & \rightarrow \ NPsg \ VPsg \\
S & \rightarrow \ NPpl \ VPpl \\
VPsg & \rightarrow \ VerbSg \ NP \\
VPpl & \rightarrow \ VerbPl \ NP \\
NPsg & \rightarrow \ DetSg \ NounSg \\
DetSg & \rightarrow \ the \ | \ a \\
\end{align*}
\]

This yields very large grammars.

What about person, case, …?

Difficult to capture generalizations

(Subject and verb have to have number agreement)

\(NPsg, \ NPpl\) and \(NP\) are three distinct nonterminals
Feature structures

Replace atomic categories with feature structures:

\[
\begin{array}{cc}
\text{CAT} & \text{NP} \\
\text{NUM} & \text{SG} \\
\text{PERS} & 3 \\
\text{CASE} & \text{NOM}
\end{array}
\quad
\begin{array}{cc}
\text{CAT} & \text{VP} \\
\text{NUM} & \text{SG} \\
\text{PERS} & 3 \\
\text{VFORM} & \text{FINITE}
\end{array}
\]

A **feature structure** is a list of **features** (= attributes, e.g. CASE), and **values** (e.g. NOM).

We often represent feature structures as **attribute value matrices (AVMs)**

Usually, values are **typed** (to avoid CASE:SG)
Feature structures as directed graphs

\[
\begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{NUM} & \text{SG} \\
\text{PERS} & 3 \\
\text{CASE} & \text{NOM}
\end{bmatrix} = \text{NP}
\]

\[
\begin{align*}
\text{CAT} & \quad \text{CASE} \\
\text{NUM} & \quad \text{NUM} \\
\text{PERS} & \quad 3 \\
\text{Nom} & \quad \text{Sg}
\end{align*}
\]
Complex feature structures

We distinguish between atomic and complex feature values.
A complex value is a feature structure itself.

This allows us to capture better generalizations.

Only atomic values:

\[
\begin{array}{c}
\text{CAT} & \text{NP} \\
\text{NUM} & \text{SG} \\
\text{PERS} & 3 \\
\text{CASE} & \text{NOM}
\end{array}
\]

Complex values:

\[
\begin{array}{c}
\text{CAT} & \text{NP} \\
\text{NUM} & \text{SG} \\
\text{PERS} & 3 \\
\text{CASE} & \text{NOM}
\end{array}
\]
A feature path allows us to identify particular values in a feature structure:

\[ \langle \text{NP CAT} \rangle = \text{NP} \]
\[ \langle \text{NP AGR CASE} \rangle = \text{NOM} \]
Unification

Two feature structures $A$ and $B$ unify ($A \sqcup B$) if they can be merged into one consistent feature structure $C$:

\[
\begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{NUM} & \text{SG} \\
\text{CASE} & \text{NOM}
\end{bmatrix} \sqcup 
\begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{PERS} & 3
\end{bmatrix} =
\begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{NUM} & \text{SG} \\
\text{PERS} & 3 \\
\text{CASE} & \text{NOM}
\end{bmatrix}
\]

Otherwise, unification fails:

\[
\begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{NUM} & \text{SG} \\
\text{CASE} & \text{NOM}
\end{bmatrix} \sqcup 
\begin{bmatrix}
\text{CAT} & \text{NP} \\
\text{NUM} & \text{PL}
\end{bmatrix} = \emptyset
\]
PATR-II style feature structures

CFG rules are augmented with constraints:

\[ A_0 \rightarrow A_1 \ldots A_n \]
\{set of constraints\}

There are two kinds of constraints:

**Unification constraints:**
\[ \langle A_i \text{ feature-path} \rangle = \langle A_j \text{ feature-path} \rangle \]

**Value constraints:**
\[ \langle A_i \text{ feature-path} \rangle = \text{atomic value} \]
# A grammar with feature structures

<table>
<thead>
<tr>
<th>S</th>
<th>→</th>
<th>NP VP</th>
<th>Grammar rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>⟨NP NUM⟩ = ⟨VP NUM⟩</td>
<td>Constraints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨NP CASE⟩ = nom</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NP</th>
<th>→</th>
<th>DT NOUN</th>
<th>Grammar rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>⟨NP NUM⟩ = ⟨NOUN NUM⟩</td>
<td>Constraints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>⟨NP CASE⟩ = ⟨NOUN CASE⟩</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOUN</th>
<th>→</th>
<th>cake</th>
<th>Lexical entry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>⟨NOUN NUM⟩ = sg</td>
<td>Constraints</td>
</tr>
</tbody>
</table>
With complex feature structures

<table>
<thead>
<tr>
<th>Grammar rule</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → NP VP</td>
<td></td>
</tr>
<tr>
<td>⟨NP AGR⟩ = ⟨VP AGR⟩</td>
<td></td>
</tr>
<tr>
<td>⟨NP CASE⟩ = nom</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grammar rule</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP → DT NOUN</td>
<td></td>
</tr>
<tr>
<td>⟨NP AGR⟩ = ⟨NOUN AGR⟩</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lexical entry</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOUN → cake</td>
<td></td>
</tr>
<tr>
<td>⟨NOUN AGR NUM⟩ = sg</td>
<td></td>
</tr>
</tbody>
</table>

Complex feature structures can capture better generalizations (and hence require fewer constraints) — cf. the previous slide
The head feature

Instead of implicitly specifying heads for each rewrite rule, let us define a head feature.

The head of a VP has the same agreement feature as the VP itself:
Re-entrancies

What we *really* want to say is that the agreement feature of the head is *identical* to that of the VP itself.

This corresponds to a *re-entrancy* in the FS (indicated via coindexation 1)

```
[ CAT VP ]
[ AGR [ NUM SG ] ]
[ HEAD [ AGR 1 ] ]
```
Re-entrancies — not like this:
Re-entrancies — but like this:
Attribute-Value Grammars and CFGs

If every feature can only have a finite set of values, any attribute-value grammar can be compiled out into a (possibly huge) context-free grammar.
Going beyond CFGs

The **power-of-2 language**: \( L_2 = \{ w^i \mid i \text{ is a power of 2} \} \)

\( L_2 \) is a (fully) context-sensitive language.

*(Mildly context-sensitive languages have the **constant growth property** (the length of words always increases by a constant factor \( c \)).)*

Here is a feature grammar which generates \( L_2 \):

\[
\begin{align*}
A & \rightarrow a \\
\langle A \; F \rangle &= 1 \\
A & \rightarrow A_1 \; A_2 \\
\langle A \; F \rangle &= \langle A_1 \rangle \\
\langle A \; F \rangle &= \langle A_2 \rangle 
\end{align*}
\]