Writing a grammar for natural language: Grammar Development

- **Grammar development** is the process of writing a grammar for a particular language.
- This can be either for a particular application or concentrating on a particular phenomena in the language under consideration.
- Check against text corpora to check the coverage of your grammar – to do this you need a parser.
- Also consider generalizations provided by a linguistic analysis.
Real Grammars get Messy

▫ Task: Capture all the morphological details which affect the syntax of a language.
▫ The CFG ends up with rules like:

\[
S \rightarrow 3\text{sgAux} 3\text{sgNP} \text{ VP}
\]
\[
S \rightarrow \text{Non3sgAux} \text{ Non3sgNP} \text{ VP}
\]
\[
3\text{sgAux} \rightarrow \text{does} | \text{has} | \text{can} | \ldots
\]
\[
\text{Non3sgAux} \rightarrow \text{do} | \text{have} | \text{can} | \ldots
\]

Real Grammars get Messy

▫ This is to deal with sentences like:
  1. Do I get dinner on this flight? (1sg = 1st person singular)
  2. Do you have a flight from Boston to Fort Worth? (2sg = 2nd person singular)
  3. Does he visit Toronto? (3sg = 3rd person singular)
  4. Does Delta fly from Atlanta to San Diego? (3sg = 3rd person singular)
  5. Do they visit Toronto? (3pl = 3rd person plural)
Real Grammars get Messy

- Not just grammatical features but also subcategorization (what kind of arguments does a verb expect?):
  
  \[
  VP \rightarrow \text{Verb-with-NP-complement } NP \quad \text{“prefer a morning flight”}
  \]
  
  \[
  VP \rightarrow \text{Verb-with-S-complement } S \quad \text{“said there were two flights”}
  \]
  
  \[
  VP \rightarrow \text{Verb-with-Inf-VP-complement } V\text{Pinf} \quad \text{“try to book a flight”}
  \]
  
  \[
  VP \rightarrow \text{Verb-with-no-complement} \quad \text{“disappear”}
  \]

Solution to non-terminal and rule blowup: Feature Structures

- **Feature structures** provide a natural way to provide complex information with each non-terminal. In some formalisms, the non-terminal is replaced with feature structures, resulting in a potentially infinite set of non-terminals.

- Feature structures are also known as f-structures, feature bundles, feature matrices, functional structures, terms (as in Prolog), or dags (directed acyclic graphs)
Feature Structures

- A feature structure is defined as a partial function from features to their values.
- For instance, we can define a function mapping the feature number onto the value singular and mapping person to third. The common notation for this function is:

  \[
  \begin{array}{l}
  \text{number: singular} \\
  \text{person: 3}
  \end{array}
  \]

Feature Structures

- Feature values can themselves be feature structures:
  
  \[
  \begin{array}{l}
  \text{cat: NP} \\
  \quad \text{agreement:}\begin{array}{l}
  \text{number: singular} \\
  \text{person: 3}
  \end{array}
  \end{array}
  \]
Feature Structures

- Consider features \( f \) and \( g \) with two distinct feature structure values of the same type:

\[
\begin{array}{c}
\text{f: } [h: a] \\
\text{g: } [h: a]
\end{array}
\]

- Feature structures can also share values. For instance, \( g \) shares the same value as \( f \) in:

\[
\begin{array}{c}
\text{f: } [\overline{1} \text{[h: a]}] \\
\text{g: } [\overline{1}]
\end{array}
\]

- The shared value is written using a co-indexation – indicating that the value is stored only once, with the index acting as a pointer.
Feature Path Notation

- The feature structure:

```
agreement: [ number: sg  ]
person: 3
subject: [ agreement: [ ] ]
```

is represented as:

```plaintext
<agreement number>=sg
<agreement person>=3
<subject agreement>=<agreement>  
```

or:

```
[ agreement = (1) [ number = 'sg', person = 3 ],
  subject = [ agreement->(1) ] ]
```

or:

```
[ agreement = ?n [ number = 'sg', person = 3 ],
  subject = [ agreement = ?n ] ]
```

Subsumption

- Feature structures have different amounts of information. Can we find an ordering on feature structures that corresponds to the compatibility and relative specificity of the information contained in them.

- **Subsumption** is a precise method of defining such an ordering over feature structures.
Subsumption

Consider the feature structure:
\[ D_{np} = \begin{bmatrix} \text{cat: NP} \end{bmatrix} \]

Compare with the feature structure:
\[ D_{np3sg} = \begin{bmatrix} \text{cat: NP} \\ \text{agreement:} \begin{bmatrix} \text{number: singular} \\ \text{person: 3} \end{bmatrix} \end{bmatrix} \]

D<sub>np</sub> makes the claim that a phrase is a noun phrase, but leaves open the question of what the agreement properties of this noun phrase are.

D<sub>np3sg</sub> also contains information about a noun phrase, but makes the agreement properties specific.

The feature structure D<sub>np</sub> is said to carry less information than, or to be more general than, or to subsume the feature structure D<sub>np3sg</sub>
Subsumption

- $D_{\text{var}} = []$
- $D_{\text{np}} = \text{[cat: NP]}$
- $D_{\text{npsg}} = \text{[cat: NP, agreement: [number: singular]}$
- $D_{\text{np3sg}} = \text{[cat: NP, agreement: [number: singular, person: 3]}$
- $D_{\text{np3sgSbj}} = \text{[cat: NP, agreement: [number: singular, person: 3, subject: [number: singular, person: 3]}$
- $D'_{\text{np3sgSbj}} = \text{[cat: NP, agreement: [number: singular, person: 3, subject: 1]}$

The following subsumption relations hold:

$D_{\text{var}} \subseteq D_{\text{np}} \subseteq D_{\text{npsg}} \subseteq D_{\text{np3sg}} \subseteq D_{\text{np3sgSbj}} \subseteq D'_{\text{np3sgSbj}}$

Unification

- Two feature structures might have different and incompatible information:
  - $\text{[cat: NP, agreement: [number: singular]}$
  - $\text{[cat: NP, agreement: [number: plural]}$
- In this case, there is no feature structure that is subsumed by both feature structures
Unification

- Subsumption is only a partial order – that is, not every two feature structures are in a subsumption relation with each other.
- Two feature structures might have different but compatible information:

  \[
  \begin{cases}
  \text{cat: NP} \\
  \text{agreement: [number: singular]} \\
  \end{cases}
  \]

  \[
  \begin{cases}
  \text{cat: NP} \\
  \text{agreement: [person: 3]} \\
  \end{cases}
  \]

Unification

- If two feature structures have different but compatible information then there always exists a more specific feature structure that is subsumed by both feature structures:

  \[
  \begin{cases}
  \text{cat: NP} \\
  \text{agreement: [number: singular]} \\
  \text{person: 3} \\
  \end{cases}
  \]
Unification

- But there are many feature structures subsumed by both of the original feature structures:

\[
\begin{array}{c}
\text{cat: NP} \\
\text{agreement:} \\
\text{number: singular} \\
\text{person: 3} \\
\text{gender: masculine}
\end{array}
\]

- So instead of considering all such feature structures we only consider the most general FS that is subsumed by the two original FSs

- This definition provides a feature structure that contains information from both input FSs but no additional information.

Unification

- Now we can define **unification**

- The *unification* of two feature structures $D'$ and $D''$ is defined as the most general feature structure $D$ such that $D' \subseteq D$ and $D'' \subseteq D$.

- This operation of unification is denoted as $D = D' \sqcup D''$
Unification

\[\square \sqcup [\text{cat: NP}] = [\text{cat: NP}]\]

\[\begin{array}{c}
\text{person: sg} \\
\text{number: 3}
\end{array}\sqcup \begin{array}{c}
\text{person: sg} \\
\text{number: 3}
\end{array} = \begin{array}{c}
\text{person: sg} \\
\text{number: 3}
\end{array}\]
Unification

[agreement: [number: sg] ] ∪ [subject: [agreement: [number: sg] ]]

[subject: [agreement: [person: 3] ]]

[agreement: [number: sg] ]

[subject: [agreement: [number: sg] ] [person: 3] ]

Unification

[agreement: [number: sg] ] ∪ [subject: [agreement: [person: 3] ]]

[subject: [agreement: [ ] ]]

[agreement: [number: sg] ] [person: 3] ]

[subject: [agreement: [ ] ] ]
Algorithms for Unification

- Represent input feature structure as a directed acyclic graph (dag). Unification is equivalent to the union-find algorithm.
- Unification is more efficient if it can be destructive: it destroys the input feature structures to create the result of unification.
- The (destructive) unification algorithm in J&M (page 423) does it in two steps: represent feature structures as dags, and then perform graph matching (and merging).
- Note that this algorithm can produce as output a dag (i.e. a feature structure) containing cycles.
  A feature structure can have part of itself as a subpart:
  \[
  \begin{array}{c}
  f: [1] \\
  g: [h: [1]]
  \end{array}
  \]
  This can be avoided with an explicit check for each call to the unify algorithm called the occur check.
- Computationally expensive since we have to traverse the whole dag at each step.

Feature Structures in CFGs

- Feature Structures impose constraints on CFG derivations:
  \[
  \begin{align*}
  S & \rightarrow NP \\
  VP & \rightarrow V NP \\
  V & \rightarrow saw \\
  NP & \rightarrow he \\
  NP & \rightarrow him \\
  NP & \rightarrow John
  \end{align*}
  \]
  This CFG derives: he saw him but not: *him saw he
  Also derives: John saw him, he saw John.
  Co-indexing in each FS is local to each CFG rule.
Feature Structures in CFGs

- A more complex example for encoding subcategorization as feature structures:

\[
S \rightarrow NP \ VP \\
\begin{array}{c}
\text{subcat: } 1 \\
\text{first: [ ]} \\
\text{rest: end}
\end{array}
\]

\[
VP \rightarrow \text{Verb}
\]

\[
VP \rightarrow \text{VP} \\
\begin{array}{c}
\text{subcat: } 1
\end{array}
\]

\[
X \rightarrow \text{cat: } 2 \ NP
\]

- In the above example, the CFG can generate an arbitrary number of NPs in the subcat feature structure for the verb.

- In effect, the above steps of unification in a CFG derivation creates a list containing the subcat elements. The subcat feature structure uses \text{first} and \text{rest} to construct the list in the recursive rule \( VP \rightarrow VP X \).

- The lexical terminal \text{Verb} can impose a constraint on which subcat frame is required.

- Other categories can be added simply by adding a new \text{cat} attribute for \( X \): e.g. \[ \text{cat: S} \] for verbs that can have a subcat of \( NP \ S \).
Unification Algorithm

function unify(f1, f2):
    returns f-structure or failure

    if f1.content == null: f1.pointer = f2
    if f2.content == null: f2.pointer = f1
    if f1.content == f2.content: f1.pointer = f2
    if f1.content and f2.content are complex f-structures:
        f2.pointer = f1
        for each f in f2.content:
            other-feature = find or create feature
                corresponding to f in f1.content
            if unify(f, other-feature) == failure:
                return failure
        return f1

Unification in Earley Parsing

- predictor: if \((A \rightarrow \alpha \bullet B \beta, [i, j], \text{dag}\_A_1)\) then \(\forall (B \rightarrow \gamma, \text{dag}\_B_1)\)
  enqueue\((B \rightarrow \bullet \gamma, [j, j], \text{dag}\_B_1), \text{chart}[j]\)  
- scanner: if \((A \rightarrow \alpha \bullet a \beta, [i, j], \text{dag}\_A_1)\) and \(a = \text{tokens}[j]\) then
  enqueue\((A \rightarrow \alpha a \bullet \beta, [i, j + 1], \text{dag}\_A_1), \text{chart}[j + 1]\)
- completer: if \((B \rightarrow \gamma \bullet, [j, k], \text{dag}\_B_1)\), for each
  \((A \rightarrow \alpha \bullet B \beta, [i, j], \text{dag}\_A_1)\)
  enqueue\((A \rightarrow \alpha B \bullet \gamma, [i, k], \text{copy-and-unify(dag}\_A_1, \text{dag}\_B_1)), \text{chart}[k]\)
  unless copy-and-unify\((\text{dag}\_A_1, \text{dag}\_B_1)\) fails
- copy-and-unify means that we make copies of the dags before
  unification because we are using a destructive unification
  algorithm
- copy-and-unify ensures that dag \(A_1\) in state
  \((A \rightarrow \alpha \bullet B \beta, [i, j], \text{dag}\_A_1)\) is not destroyed since it can be
  used in the completer with other states and unify with them.
Unification in Earley Parsing

- Consider two different enqueue requests:
  enqueue((A → α B • γ, [i, k], dag_{A_1}), chart[k])
  enqueue((A → α B • γ, [i, k], dag_{A_2}), chart[k])

- Consider the case where:
  \(\text{dag}_{A_1} = [\text{tense: past | plural}]\) and
  \(\text{dag}_{A_2} = [\text{tense: past}]\)
  Clearly, \(\text{dag}_{A_1} \sqsubseteq \text{dag}_{A_2}\)

Unification in Earley Parsing

- Which feature structure should be selected after the two enqueue commands above?
  Three options: \(\text{dag}_{A_1}, \text{dag}_{A_2}, \text{dag}_{A_1} \sqcup \text{dag}_{A_2}\)

- In general, the feature inserted should subsume both \(\text{dag}_{A_1}\) and \(\text{dag}_{A_2}\)

- In practice exactly one of the following conditions is always true:
  - If \(\text{dag}_{A_1} \sqsubseteq \text{dag}_{A_2}\) then enqueue picks \(\text{dag}_{A_1}\),
  - If \(\text{dag}_{A_2} \sqsubseteq \text{dag}_{A_1}\) then enqueue picks \(\text{dag}_{A_2}\),
  - If \(\text{dag}_{A_1} \not\sqsubseteq \text{dag}_{A_2}\) and \(\text{dag}_{A_2} \not\sqsubseteq \text{dag}_{A_1}\) then enqueue picks \(\text{dag}_{A_1} \sqcup \text{dag}_{A_2}\)
Unification in Earley Parsing

- During the enqueue of a state, we always pick the most general feature structure possible.
- To see why consider an example:
  - Consider a chart which contains the state:
    \[ S_1 = (NP \rightarrow \cdot DT \, NP, [i, i], \text{dag}_{S_1} = []) \]
  - The parser then tries to enqueue a new state:
    \[ S_2 = (NP \rightarrow \cdot DT \, NP, [i, i], \text{dag}_{S_2} = [\text{DT.num} = \text{sing}]) \]
  - Consider two possible situations:
    1. a singular DT is scanned, then either \( \text{dag}_{S_1} \) or \( \text{dag}_{S_2} \) would unify and parsing would continue.
    2. a plural DT is scanned, then if we picked \( \text{dag}_{S_2} \) we have a unification failure; on the other hand picking the more general feature structure \( \text{dag}_{S_1} \) allows parsing to continue.

- So, if there are two possible ways to derive a span, then the most general feature structure is the one we must choose.

Summary

- Feature structures generalize the notion of non-terminals in a grammar.
- Complex morphological details can be encoded into a feature structure.
- Feature structures can have shared or co-referential parts.
- Feature structures can implement arbitrary lists (the notation is very computationally powerful).
- Unification provides a means to combine the information in two feature structures.
- Feature structures can be used in a context-free grammar, and
- Unification is done while parsing to ensure that the constraints specified in the features are not violated.