Idea 1

Transform into CFG with atomic categories

If only compound terms without variables are used, the rules correspond to rules with atomic categories

Example:
- $S \rightarrow NP_{1,sg} \ VP_{1,sg}$
- $S \rightarrow NP_{1,pl} \ VP_{1,pl}$

More on Idea 1

If there are a finite set of possible values for the variables occurring in the compound terms, one can replace a rule with the instances for all possible instantiations of variables

Example:
- $S \rightarrow NP(1,sg) \ VP(1,sg)$
- $S \rightarrow NP(1,pl) \ VP(1,pl)$
- $S \rightarrow NP(2,sg) \ VP(2,sg)$
- $S \rightarrow NP(2,pl) \ VP(2,pl)$
- $S \rightarrow NP(3,sg) \ VP(3,sg)$
- $S \rightarrow NP(3,pl) \ VP(3,pl)$

Evaluation of Idea 1

- Leads to a potentially huge set of rules
  - number of categories grows exponentially w.r.t. the number of features
  - grammar size relevant for time & space efficiency of parsing
- Doesn’t allow for variables, i.e., misses generalizations
Feature structures for parsing
Idea 2
Parse using atomic CFG backbone (reduced info)

- Idea:
  - parse using a property defined for all categories
  - use other properties to filter solutions from set of parses
- Downside:
  - parsing with partial information can significantly enlarge the search space

Feature structures for parsing
Idea 3
Incorporate special mechanism into parser

- How two categories are combined has to be replaced by unification.
- Every active and inactive edge in a chart may be used for different uses.
  - So, for each time an edge is used, a new copy needs to be made.
- Two effectiveness issues:
  - Use subsumption test to ensure general enough predictions
  - Use restriction to prevent prediction loops
- Two efficiency issues (not dealt with here):
  - intelligent indexing of edges in chart
  - packing of similar edges in chart (cf., Tomita parser)

Exploring Unification

Taking idea 3, here’s where we’re going:
- Feature Structures
- Unification
- Unification-Based Grammars
- Chart Parsing with Unification-Based Grammars
(next slide set)

Feature structures

To address the problem of adding agreement to CFGs, we need features, e.g., a way to say:

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>sg</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERSON</td>
<td>3</td>
</tr>
</tbody>
</table>

A structure like this allows us to state properties, e.g., about a noun phrase:

<table>
<thead>
<tr>
<th>CAT</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>sg</td>
</tr>
<tr>
<td>PERSON</td>
<td>3</td>
</tr>
</tbody>
</table>

- Each feature (e.g., NUMBER) is paired with a value (e.g., sg)
  - A bundle of feature-value pairs can be put into an attribute-value matrix (AVM)

Feature paths

Values can be atomic (e.g. sg or NP or 3):

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>sg</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERSON</td>
<td>3</td>
</tr>
</tbody>
</table>

Or they can be complex, allowing for feature paths:

<table>
<thead>
<tr>
<th>CAT</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGREEMENT</td>
<td>NUMBER</td>
</tr>
<tr>
<td></td>
<td>PERSON</td>
</tr>
</tbody>
</table>

The value of the path [AGREEMENT | NUMBER] is sg

- Complex values allow for more expressivity than a CFG, i.e., can represent more linguistic phenomena

Constraints

Idea: each rule of the grammar is a complex bundle of constraints

- S → NP VP means that an S object is constrained to be composed of an NP followed by a VP

Features allow one to add more constraints

- S → NP VP only if number of NP = number of VP
  - Constraint 1: S → NP VP
  - Constraint 2: NP.num = VP.num

Often referred to as constraint-based processing
Feature structures as graphs

- Feature structures are directed acyclic graphs (DAGs).
- The feature structure represented by the attribute-value matrix (AVM):

\[
\begin{array}{c}
\text{CAT} \\
\text{NP} \\
\text{AGR} \\
\text{NUM} \\
\text{SG} \\
\text{PER} \\
\text{3}
\end{array}
\]

is really the graph:

\[
\begin{array}{c}
\text{NP} \\
\text{sg} \\
\text{AGR} \\
\text{num} \\
\text{sg} \\
\text{per} \\
\text{3}
\end{array}
\]

Reentrancy (structure sharing)

Feature structures embedded in feature structures can share the same values:

- Two features share precisely the same object as their value.
  - We'll indicate this with a tag like \[
  \begin{array}{c}
  \text{cat} \\
  \text{S} \\
  \text{HEAD} \\
  \text{AGR} \\
  \text{num} \\
  \text{sg} \\
  \text{per} \\
  \text{3}
  \end{array}
  \]
  - This is really the graph:

\[
\begin{array}{c}
\text{NP} \\
\text{subj} \\
\text{agr} \\
\text{num} \\
\text{sg} \\
\text{per} \\
\text{3}
\end{array}
\]

What structure-sharing is not

- This is structure-sharing (changing value in one place changes both):

\[
\begin{array}{c}
\text{HEAD} \\
\text{AGR} \\
\text{num} \\
\text{sg} \\
\text{per} \\
\text{3}
\end{array}
\]

- This is not (changing one value doesn’t change other):

\[
\begin{array}{c}
\text{HEAD} \\
\text{AGR} \\
\text{num} \\
\text{sg} \\
\text{per} \\
\text{3}
\end{array}
\]

Unification

We'll often want to merge feature structures:

- **Unification** ($\sqcup$) = a basic operation to merge two feature structures into a resultant feature structure (FS).

The two feature structures must be compatible, i.e., have no values that conflict:

- Identical FSs:
  \[
  \text{number sg} \sqcup \text{number sg} = \text{number sg}
  \]

- Conflicting FSs:
  \[
  \text{number sg} \sqcup \text{number pl} = \text{Fail}
  \]

- Merging with an unspecified FS:
  \[
  \text{number sg} \sqcup \text{number sg} = \text{number sg}
  \]

Unification (cont.)

- Merging FSs with different features specified:
  \[
  \text{number sg} \sqcup \text{person 3} = \text{number sg}
  \]

- More examples:

\[
\begin{array}{c}
\text{CAT} \sqcup \text{NP} \sqcup \text{AGR} \sqcup \text{num} \sqcup \text{sg} = \text{CAT} \sqcup \text{NP} \sqcup \text{AGR} \sqcup \text{num} \sqcup \text{sg}
\end{array}
\]

Unification with Reentrancies

- Remember that structure-sharing means they are the same object:

\[
\begin{array}{c}
\text{AGR} \sqcup \text{per} \sqcup \text{num} \sqcup \text{sg} = \text{AGR} \sqcup \text{per} \sqcup \text{num} \sqcup \text{sg}
\end{array}
\]

- When unification takes place, shared values are copied over:

\[
\begin{array}{c}
\text{AGR} \sqcup \text{per} \sqcup \text{num} \sqcup \text{sg} = \text{AGR} \sqcup \text{per} \sqcup \text{num} \sqcup \text{sg}
\end{array}
\]
Unification with Reentrancies (cont.)

- And remember that having similar values is not the same as structure-sharing:
  \[ \begin{array}{c}
  \text{AGR} \\
  \text{NUM} \\
  \text{SG}
  \end{array} \] 
  \[ \begin{array}{c}
  \text{SUBJ} \\
  \text{AGR} \\
  \text{NUM} \\
  \text{SG}
  \end{array} \] 
  \[ \begin{array}{c}
  \text{SUBJ} \\
  \text{AGR} \\
  \text{PER} \\
  \text{3}
  \end{array} \] 

- With structure-sharing, the values must be compatible everywhere it is specified:
  \[ \begin{array}{c}
  \text{AGR} \\
  \text{NUM} \\
  \text{SG}
  \end{array} \] 
  \[ \begin{array}{c}
  \text{SUBJ} \\
  \text{AGR} \\
  \text{NUM} \\
  \text{SG}
  \end{array} \] 
  \[ \begin{array}{c}
  \text{SUBJ} \\
  \text{AGR} \\
  \text{PER} \\
  \text{3}
  \end{array} \] 

Subsumption

A more general feature structure (less values specified) subsumes a more specific feature structure:

1. \[ \text{NUM} \text{ SG} \]
2. \[ \text{PER} \text{ 3} \]
3. \[ \text{NUM} \text{ SG} \]

The following subsumption relations hold:

- (1) subsumes (3)
- (2) subsumes (3)
- (1) does not subsume (2), and (2) does not subsume (1)

Implementing Unification

How do we implement a check on unification?

- **Goal**: given feature structures F1 and F2, return F, the unification of F1 and F2

Unification is a recursive operation:

- If a feature has an atomic value, see if the other FS has that feature with the same value
  - \[ F \text{ a} \] unifies with \[ F \text{ a} \]

- If a feature has a complex value, follow the paths to see if they're compatible & have the same values at bottom
  - To see whether \[ F \text{ G1} \] unifies with \[ F \text{ G2} \], inspect G1 and G2

- To avoid cycles, do an **occur check** to see if we’ve seen a FS before or not

Unification-based grammars

Grammars with feature structures

One way to encode features is to augment a CFG skeleton with feature structure path equations

- CFG skeleton
  \[ S \rightarrow \text{NP} \text{ VP} \]

- Path equations
  \[ (\text{NP AGREEMENT}) = (\text{VP AGREEMENT}) \]

Conditions:

1. There can be zero or more path equations for each rule skeleton — no longer atomic
2. When a path equation references constituents, they can only be constituents from the CFG rule

The need for unification

Assume:

- a verb selecting for a 3rd person singular noun subject
- a subject which is 2nd person singular

What the verb specifies for the subject has to be able to unify with what the subject is

- In this case, unification will fail: person doesn’t unify

Handling Linguistic Phenomena

We’ll look at 3 different phenomena that feature-based, or unification-based, grammars capture fairly succinctly:

1. Agreement
2. Subcategorization
3. Long-distance dependencies

You can find our more details by exploring:

- Lexical-Functional Grammar (LFG)
- Head-driven Phrase Structure Grammar (HPSG)
(Both are taught in Alternative Syntactic Theories (L614))
1) Agreement in Feature-based Grammars

One way to capture agreement rules:

- **S → NP VP**
  - \((S \text{ head}) = (V \text{ head})\)
  - \((NP \text{ head agr}) = (V \text{ head agr})\)

- **VP → V NP**
  - \((V \text{ head}) = (V \text{ head})\)

- **NP → D Nom(inal)**
  - \((NP \text{ head}) = (Nom \text{ head})\)
  - \((Det \text{ head agr}) = (Nom \text{ head agr})\)

- **Nom → Noun**
  - \((Nom \text{ head}) = (Noun \text{ head})\)

- **Noun → flights**
  - \((Noun \text{ head agr num}) = pl\)

2) Subcategorization

We could specify subcategorization like so:

- **VP → V NP**
  - \((V \text{ subcat}) = \text{intrans}\)
- **VP → V NP**
  - \((V \text{ subcat}) = \text{trans}\)
- **VP → V NP**
  - \((V \text{ subcat}) = \text{ditrans}\)

But values like \text{intrans} do not correspond to anything that the rules actually look like:

- To make \text{subcat} better match the rules, we can make its value a list of a verb's arguments, e.g., \langle NP, PP, \rangle

More formal (Prolog-esque) way to specify lists:

\[
\langle \text{NP}, \text{PP} \rangle \text{ is equivalent to:}
\begin{array}{c}
\text{FIRST} \\
\text{REST} \\
\text{FIRST} \\
\text{REST} \\
\text{REST} \\
\end{array}
\]
Handling Subcategorization

How do we ensure that an object’s subcategorization list corresponds to what we see in the actual tree?

- We need a subcategorization principle

As a tree is built, items are checked off of the subcat list

- The subcat list must be empty at the top of a tree
- If we had used the rule VP → V NP, we would have been left with subcat <NP,PP>
- The rule VP → V NP PP PP would have specified something missing from the subcat list

Handling long-distance dependencies

TOP:

(fill gap) S → wh-word be-cop NP
(NP gap) = (wh-word head)

MIDDLE:

(pass gap) NP → D Nom
(Nom gap) = (Nom gap)
Nom → Nom RelCl
(Nom gap) = (RelCl gap)
RelCl → RelPro NP VP
(RelCl gap) = (VP gap)

BOTTOM:

(identify gap) VP → V
(VP gap) ∈ (V subcat)

(Actually, we want a more general principle to introduce gap features, but this will do for now …)

What’s going on

- Traces, or gaps, are allowed as items from subcat lists
- When a trace is introduced, make sure it gets checked off subcat, so the subcat principle is satisfied
- Alternate way: the gap value of a mother of a rule is the union of the daughter’s gap values
  - So, we wouldn’t have to write separate rules for RelClause, Nom, NP, etc.
  - When a subcat list is empty & an item matches something in the gap set, remove it from gap

3) Long-distance dependencies

Long-distance dependencies are often also called “movement” phenomena

- Topicalization: John she likes …
- Wh-questions: Who does she like …?

To capture this without movement, one can instead pass features along the tree

- Bottom: introduce a ‘trace’
- Middle: pass the trace
- Top: Unify the features of the trace with some real word (e.g., John, Who)

We’ll use a gap feature for this