Topics in Computational Linguistics
— Grammar Engineering —

Dan Flickinger
Stephan Oepen

Center for the Study of Language and Information
Stanford University — Winter Quarter 2003

{dan|oe}@csli.stanford.edu
Outline: What We Are About to Do (and Why)

Course Outline

- familiarize with computational grammar development environment;
- learn how to formalize grammars in typed feature structures;
- adapt and develop sequence of trivial HPSG grammars in LKB;
- solve weekly exercises: immediate gratification (risk of late hours).

Why Computational Grammars

- **research** formalize linguistic theories with complex interactions of language phenomena; identify cross-language generalizations;
- **education** teach frameworks or analyses in formal morphology, syntax, and semantics; support student experimentation;
- **applications** embed grammar-based natural language analysis in research prototypes and commercial applications.
The Type Hierarchy: Fundamentals

- Types ‘represent’ groups of entities with similar properties (‘classes’);
- types ordered by specificity: subtypes inherit properties of (all) parents;
- type hierarchy determines which types are compatible (and which not).
Properties of (Our) Type Hierarchies

- **Unique Top** a single hierarchy of all types with a unique top node;
- **No Cycles** no path through the hierarchy from one type to itself;
- **Unique Greatest Lower Bounds** Any two types in the hierarchy are either (a) incompatible (i.e. share no descendants) or (b) have a unique most general (‘highest’) descendant (called their greatest lower bound);
- **Closed World** all types that exist have a known position in hierarchy;
- **Compatibility** type compatibility in the hierarchy determines feature structure unifiability: two types unify to their greatest lower bound.
Multiple Inheritance

- flyer and swimmer have no common descendants: they are incompatible;
- flyer and bee stand in hierarchical relationship: they unify to subtype;
- flyer and invertebrate have a unique greatest common descendant.

Diagram:

```
*top*

animal
  /   
flyer swimmer invertebrate vertebrate
  /  /     
bee bee fish
  /  /    
  cod guppy
```
An Invalid Type Hierarchy

- swimmer and vertebrate have two joint descendants: fish and whale;
- fish and whale are incomparable in the hierarchy: glb condition violated.
• LKB system introduces glb types as required: ‘swimmer-vertebrate’.

Diagram:

- *top*
- animal
  - flyer
  - swimmer
  - invertebrate
  - vertebrate
    - glbtype42
      - fish
        - cod
        - guppy
      - whale
      - mammal
        - dog

Grammar Engineering (7)
Typed Feature Structures (as Graph)

phrase → verb

ARGS

*ne-list*

FIRST

ORTH "chased"

HEAD

verb

REST

*ne-list*

FIRST

syn-struc

HEAD

noun

REST

*null*
Properties of Typed Feature Structures

- **Finiteness**  a typed feature structure has a finite number of nodes;

- **Unique Root and Connectedness**  a typed feature structure has a unique root node; apart from the root, all nodes have at least one parent;

- **No Cycles**  no node has an arc that points back to the root node or to another node that intervenes between the node itself and the root;

- **Unique Features**  any node can have any (finite) number of outgoing arcs, but the arc labels (i.e. features) must be unique within each node;

- **Typing**  each node has single type which is defined in the hierarchy.
Our Example Structure as an AVM
Our Example Structure in the Description Language

\[
\text{foo} := \text{phrase} \& \\
[ \ \text{HEAD} \ \text{verb}, \\
\ \text{ARGS} \ \ast\text{ne-list}\ast \& \\
[ \ \text{FIRST} \ \text{word} \& \\
[ \ \text{ORTH} \ "chased", \\
\ \text{HEAD} \ \text{verb} ], \\
\ \text{REST} \ \ast\text{ne-list}\ast \& \\
[ \ \text{FIRST} \ \text{syn-struc} \& \\
[ \ \text{HEAD} \ \text{noun} ], \\
\ \text{REST} \ \ast\text{null}\ast \ ]] ] .
\]
Reentrancy in a Typed Feature Structure (Graph)
Reentrancy in a Typed Feature Structure (AVM)
Reentrancy in a Typed Feature Structure (TDL)

bar := phrase &
[ HEAD #head & verb,
 ARGS *ne-list* &
 [ FIRST word &
   [ ORTH "chased",
     HEAD #head ],
   REST *ne-list* &
   [ FIRST syn-struc &
     [ HEAD noun ],
     REST *null* ]]] .
Typed Feature Structure Subsumption

- Typed feature structures can be partially ordered by information content;
- a more general structure is said to subsume a more specific one;
- *top* is the most general feature structure (while ⊥ is inconsistent);
- ⊑ (‘square subset or equal’) conventionally used to depict subsumption.

Feature structure $F$ subsumes feature structure $G$ ($F ⊑ G$) iff: (1) if path $p$ is defined in $F$ then $p$ is also defined in $G$ and the type of the value of $p$ in $F$ is a supertype or equal to the type of the value of $p$ in $G$, and (2) all paths that are reentrant in $F$ are also reentrant in $G$. 
Feature Structure Subsumption: Examples

Feature structure $F$ subsumes feature structure $G$ ($F \sqsubseteq G$) iff: (1) if path $p$ is defined in $F$ then $p$ is also defined in $G$ and the type of the value of $p$ in $F$ is a supertype or equal to the type of the value of $p$ in $G$, and (2) all paths that are reentrant in $F$ are also reentrant in $G$. 
Typed Feature Structure Unification

- Decide whether two typed feature structures are mutually compatible;
- determine combination of two TFSs to give the most general feature structure which retains all information which they individually contain;
- if there is no such feature structure, unification fails (depicted as ⊥);
- unification monotonically combines information from both ‘input’ TFSs;
- relation to subsumption the unification of two structures $F$ and $G$ is the most general TFS which is subsumed by both $F$ and $G$ (if it exists).
- $\sqcap$ (‘square set intersection’) conventionally used to depict unification.
Typed Feature Structure Unification: Examples

TFS\(_1\): \[
\begin{bmatrix}
\text{FOO} & x \\
\text{BAR} & x
\end{bmatrix}
\]

TFS\(_2\): \[
\begin{bmatrix}
\text{FOO} & x \\
\text{BAR} & y
\end{bmatrix}
\]

TFS\(_3\): \[
\begin{bmatrix}
\text{FOO} & y \\
\text{BAR} & x \\
\text{BAZ} & x
\end{bmatrix}
\]

TFS\(_4\): \[
\begin{bmatrix}
\text{FOO} & 1 & x \\
\text{BAR} & 1
\end{bmatrix}
\]

Signature

\[
\begin{align*}
\text{Signature} &= \\
P_{1} &\quad \text{FOO} \quad \text{BAR} \\
P_{2} &\quad \text{FOO} \quad \text{BAR} \\
P_{3} &\quad \text{FOO} \quad \text{BAR} \\
P_{4} &\quad \text{FOO} \quad \text{BAR} \\
\end{align*}
\]

TFS\(_1 \sqcap TFS\(_2 \equiv TFS\(_2 \)

TFS\(_1 \sqcap TFS\(_3 \equiv TFS\(_3 \)

TFS\(_3 \sqcap TFS\(_4 \equiv TFS\(_3 \)

TFS\(_3 \sqcap TFS\(_4 \equiv TFS\(_3 \)

Grammar Engineering (18)
Type Constraints and Appropriate Features

- Well-formed TFSs satisfy all type constraints from the type hierarchy;
- type constraints are typed feature structures associated with a type;
- the top-level features of a type constraint are appropriate features;
- type constraints express generalizations over a ‘class’ (set) of objects.

<table>
<thead>
<tr>
<th>type</th>
<th>constraint</th>
<th>appropriate features</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ne-list</em></td>
<td>FIRST <em>top</em></td>
<td>FIRST and REST</td>
</tr>
<tr>
<td></td>
<td>REST <em>list</em></td>
<td></td>
</tr>
</tbody>
</table>
Type Inference: Making a TFS Well-Formed

• Apply all type constraints to convert a TFS into a well-formed TFS;
• determine most general well-formed TFS subsumed by the input TFS;
• specialize all types so that all features are appropriate:

\[
\begin{pmatrix}
    \text{HEAD} & \text{pos} \\
    \text{ARGS} & \text{*list*}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
    \text{HEAD} & \text{pos} \\
    \text{ARGS} & \text{*list*}
\end{pmatrix}
\]

• expand all nodes with the type constraint of the type of that node:

\[
\begin{pmatrix}
    \text{HEAD} & \text{pos} \\
    \text{ARGS} & \text{*list*}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
    \text{HEAD} & \text{pos} \\
    \text{ARGS} & \text{*list*} \\
    \text{COMPS} & \text{*list*}
\end{pmatrix}
\]
More Interesting Well-Formed Unification

Type Constraints Associated to Earlier animal Hierarchy

\[
\begin{align*}
\text{swimmer} & \rightarrow \text{swimmer} \left[ \begin{array}{c} \text{FINS bool} \\ \text{true} \end{array} \right] \\
\text{mammal} & \rightarrow \text{mammal} \left[ \begin{array}{c} \text{FRIENDLY bool} \\ \text{true} \end{array} \right] \\
\text{whale} & \rightarrow \text{whale} \left[ \begin{array}{c} \text{BALEEN bool} \\ \text{true} \\ \text{FINS true} \\ \text{FRIENDLY bool} \end{array} \right] \\
\end{align*}
\]

\[
\begin{align*}
\text{mammal} \left[ \begin{array}{c} \text{FRIENDLY true} \end{array} \right] & \sqcap \text{swimmer} \left[ \begin{array}{c} \text{FINS bool} \end{array} \right] \equiv \text{whale} \left[ \begin{array}{c} \text{BALEEN bool} \\ \text{true} \\ \text{FINS true} \\ \text{FRIENDLY true} \end{array} \right] \\
\text{mammal} \left[ \begin{array}{c} \text{FRIENDLY true} \end{array} \right] & \sqcap \text{swimmer} \left[ \begin{array}{c} \text{FINS false} \end{array} \right] \equiv \bot \\
\end{align*}
\]
Recursion in the Type Hierarchy

• Type hierarchy must be finite after type inference; illegal type constraint:

  *list* := *top* & [ FIRST *top*, REST *list* ].

• needs additional provision for empty lists; indirect recursion:

  *list* := *top*.
  *ne-list* := *list* & [ FIRST *top*, REST *list* ].
  *null* := *list*.

• recursive types allow for parameterized list types (‘list of X’):

  *s-list* := *list*.
  *s-ne-list* := *ne-list* & *s-list &
  [ FIRST syn-struc, REST *s-list* ].
  *s-null* := *null* & *s-list*. 
Notational Conventions

• lists not available as built-in data type; abbreviatory notation in TDL:
  \(< a, b > \equiv [ \text{FIRST } a, \text{REST } [ \text{FIRST } b, \text{REST } *\text{null}* ] ]\)

• underspecified (variable-length) list:
  \(< a \ldots > \equiv [ \text{FIRST } a, \text{REST } *\text{list} ]\)

• difference (open-ended) lists; allow concatenation by unification:
  \(<! a !> \equiv [ \text{LIST } [ \text{FIRST } a, \text{REST } #\text{tail} ], \text{LAST } #\text{tail} ]\)

• built-in and ‘non-linguistic’ types pre- and suffixed by asterisk (*\text{top}*);
• strings (e.g. “\text{chased}”) need not be declared; always subtypes of string;
• strings cannot have subtypes and are (thus) mutually incompatible.
Structured Categories in a Unification Grammar

- All (constituent) categories in the grammar are typed feature structures;
- specific TFS configurations may correspond to ‘traditional’ categories;
- labels like ‘S’ or ‘NP’ are mere abbreviations, not elements of the theory.
The Format of Grammar Rules in the LKB

mother

\[
\begin{bmatrix}
\text{HEAD} & 1 \\
\text{SPR} & 2 \\
\text{COMPS} & \langle \rangle \\
\cdots
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\text{HEAD} & 1 \\
\text{SPR} & 2 \\
\text{COMPS} & \langle 3 \rangle \\
\end{bmatrix},
\begin{bmatrix}
\text{ARGS} & \langle \rangle \\
\end{bmatrix}
\begin{bmatrix}
\text{daughter}_1 \\
\end{bmatrix}
\]

\begin{bmatrix}
\text{HEAD} & 1 \\
\text{SPR} & 2 \\
\text{COMPS} & \langle 3 \rangle \\
\end{bmatrix},
\begin{bmatrix}
\text{daughter}_2 \\
\end{bmatrix}
\]
Our Grammars: Table of Contents

## Type Description Language (TDL)

- **types.tdl**  
  type definitions: hierarchy of grammatical knowledge;

- **lexicon.tdl**  
  instances of (lexical) types plus orthography;

- **rules.tdl**  
  instances of construction types; used by the parser;

- **lrules.tdl**  
  lexical rules, applied before non-lexical rules;

- **irules.tdl**  
  lexical rules that require orthographemic variation.

## Auxiliary Files (Grammar Configuration for LKB)

- **globals.lsp**  
  Parameter settings (e.g. path to orthography et al.);

- **user-fns.lsp**  
  (small number) of LKB interface functions;

- **mrsglobals.lsp**  
  MRS parameters (path to semantics et al.)