Precision Grammar Implementation for Linguistic Hypothesis Testing

Emily M. Bender
University of Washington
ebender@u.washington.edu

Dan Flickinger
CSLI Stanford
danf@csli.stanford.edu

Stephan Oepen
Universitetet i Oslo & CSLI Stanford
oe@csli.stanford.edu

http://lingo.stanford.edu/courses/07/lsa/
The Linguistic Knowledge Builder (LKB)

General & History
- Specialized grammar engineering environment for TFS grammars;
- main developers: Copestake (original), Carroll, Malouf, and Oepen;
- open-source and binary distributions (Linux, Windows, and Solaris).

Grammar Engineering Functionality
- Compiler for typed feature structure grammars $\rightarrow$ wellformedness;
- parser and generator: map from strings to meaning and vice versa;
- visualization: inspect trees, feature structures, intermediate results;
Recognizing the Language of a Grammar \( \langle C, \Sigma, P, S \rangle \)

\[
P \equiv \\
S \rightarrow \text{NP VP} \\
\text{VP} \rightarrow \text{V NP} \\
\text{VP} \rightarrow \text{VP PP} \\
\text{NP} \rightarrow \text{NP PP} \\
\text{PP} \rightarrow \text{P NP} \\
\text{NP} \rightarrow \text{kim | snow | oslo} \\
\text{V} \rightarrow \text{snores | adores} \\
\text{P} \rightarrow \text{in}
\]

All Complete Derivations

- are rooted in the start symbol \( S \);
- label internal nodes with categories \( \in C \), leaves with words \( \in \Sigma \);
- instantiate a grammar rule \( \in P \) at each local subtree of depth one.

\[\text{S} \]

\[\begin{array}{c}
\text{NP} \\
\mid \text{kim} \\
\text{VP} \\
\mid \text{V} \\
\mid \text{NP} \\
\text{PP} \\
\mid \\
\text{P} \\
\text{NP} \\
\mid \text{adores} \\
\mid \text{snow} \\
\mid \text{in} \\
\mid \text{oslo} \\
\end{array}\]
Limitations of Context-Free Grammar

Agreement and Valency (For Example)

That dog barks.
*That dogs barks.
*Those dogs barks.
The dog chased a cat.
*The dog barked a cat.
*The dog chased.
*The dog chased a cat my neighbours.
The cat was chased by a dog.
*The cat was chased of a dog.
...
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.

```
word  HEAD noun
      SPR  [HEAD det]
      COMPS  []

phrase HEAD verb
      SPR  [HEAD noun]
      COMPS  []

phrase HEAD verb
      SPR  [HEAD noun]
      COMPS  []

phrase HEAD verb
      SPR  [HEAD noun]
      COMPS  []

phrase HEAD verb
      SPR  [HEAD noun]
      COMPS  []
```

‘N’ ‘lexical’

‘S’ ‘maximal’

‘VP’ ‘intermediate’
Interaction of Lexicon and Phrase Structure Schemata

```
[HEAD 1]
[SPR 〈〉]
[COMPS 3]  →  [SPR 〈〉]
phrase

[ORTH "Kim"]
[HEAD noun]
[AGR 3sg]
[SPR 〈〉]
[COMPS 〈〉]

[ORTH "sleeps"]
[HEAD verb]
[AGR 1 3sg]
[SPR 〈〉]
[COMPS 〈〉]

[ORTH 〈〉]
[HEAD noun]
[AGR 1]
[SPR 〈〉]
[COMPS 〈〉]
```
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).
Multiple Inheritance

- *flyer* and *swimmer* 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.
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 $\bot$ is inconsistent);
- $\sqsubseteq$ (‘square subset or equal’) conventionally used to depict subsumption.

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 $G$.
Feature structure $F$ subsumes feature structure $G$ ($F \subseteq G$) 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 $\perp$);
- 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).
- $\Box$ (‘square set intersection’) conventionally used to depict unification.
Typed Feature Structure Unification: Examples

TFS₁: \[
\begin{array}{c}
  FOO \ x \\
  BAR \ x \\
\end{array}
\]
\(a\)

TFS₂: \[
\begin{array}{c}
  FOO \ x \\
  BAR \ y \\
\end{array}
\]
\(a\)

TFS₃: \[
\begin{array}{c}
  FOO \ y \\
  BAR \ x \\
  BAZ \ x \\
\end{array}
\]
\(b\)

TFS₄: \[
\begin{array}{c}
  FOO \ \boxed{1} \ x \\
  BAR \ \boxed{1} \\
\end{array}
\]
\(a\)

Signature

\[
\begin{array}{c}
  a \quad FOO \\
  BAR \\
\end{array}
\]
\(x\)

\[
\begin{array}{c}
  b \quad BAZ \\
\end{array}
\]
\(y\)

TFS₁ \(\sqcap\) TFS₂ \(\equiv\) TFS₂
TFS₁ \(\sqcap\) TFS₃ \(\equiv\) TFS₃
TFS₃ \(\sqcap\) TFS₄ \(\equiv\)
\[
\begin{array}{c}
  FOO \ \boxed{1} \ y \\
  BAR \ \boxed{1} \\
  BAZ \ x \\
\end{array}
\]
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>
</table>
| *ne-list*          | $\begin{bmatrix}
                          \text{FIRST} & \text{*top*} \\
                          \text{REST} & \text{*list*}
                     \end{bmatrix}$ | FIRST and REST       |
Recursion in the Type Hierarchy

- Type hierarchy must be finite after type inference; illegal type constraint:
  \[ \text{*list*} := \text{*top*} \& \left[ \text{FIRST} \text{*top*}, \text{REST} \text{*list*} \right]. \]

- needs additional provision for empty lists; indirect recursion:
  \[ \begin{align*}
  \text{*list*} & := \text{*top*}. \\
  \text{*ne-list*} & := \text{*list*} \& \left[ \text{FIRST} \text{*top*}, \text{REST} \text{*list*} \right]. \\
  \text{*null*} & := \text{*list*}. 
  \end{align*} \]

- recursive types allow for parameterized list types (‘list of X’):
  \[ \begin{align*}
  \text{*s-list*} & := \text{*list*}. \\
  \text{*s-ne-list*} & := \text{*ne-list*} \& \text{*s-list*} \& \\
  & \left[ \text{FIRST} \text{expression}, \text{REST} \text{*s-list*} \right]. \\
  \text{*s-null*} & := \text{*null*} \& \text{*s-list*}. 
  \end{align*} \]
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 (*top*);
- strings (e.g. “chased”) need no declaration; always subtypes of *string*;
- strings cannot have subtypes and are (thus) mutually incompatible.