Computational Linguistics II
— Grammars, Algorithms, Statistics —

Dan Flickinger

Oslo and Stanford Universities

danf@csli.stanford.edu

Tore Langholm

Universitetet i Oslo

torel@ifi.uio.no

Stephan Oepen

Oslo and Stanford Universities

oe@csli.stanford.edu

The Linguistic Knowledge Builder (LKB)

General & History

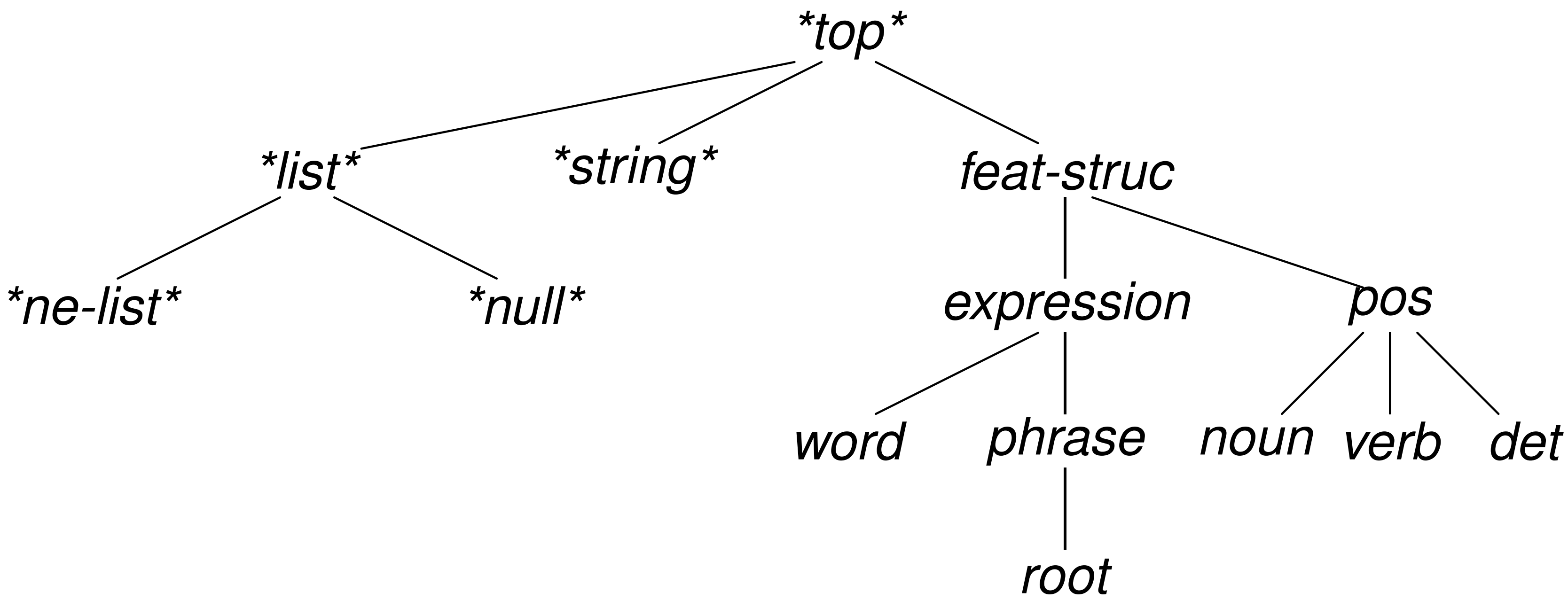
- Specialized grammar engineering environment for TFS grammars;
- main developers: Copestake (original), Carroll, Malouf, and Oepen;
- open-source and binary distributions (Linux, Windows, and Solaris).

Grammar Engineering Fuctionality

- 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;
- debugging and tracing: interactive unification, ‘stepping’, et al.

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).

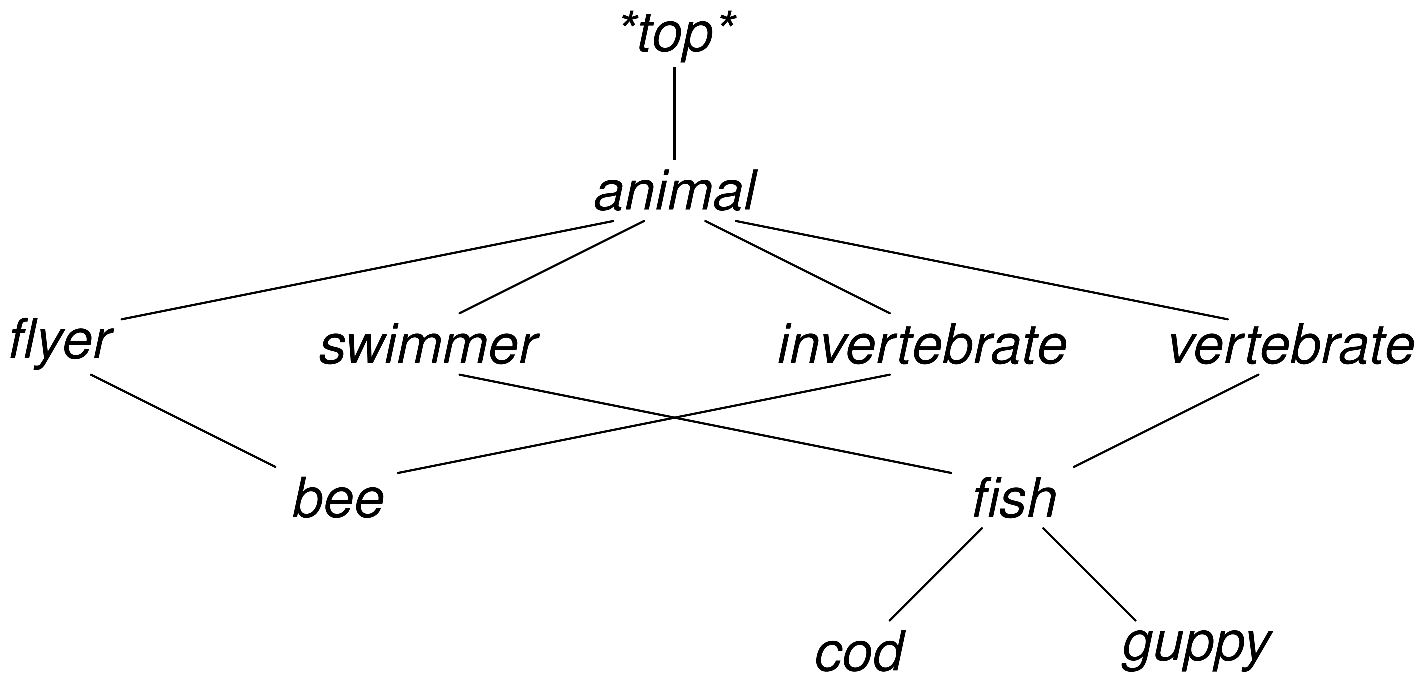


Properties of (Our) Type Hierarchies

- **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.

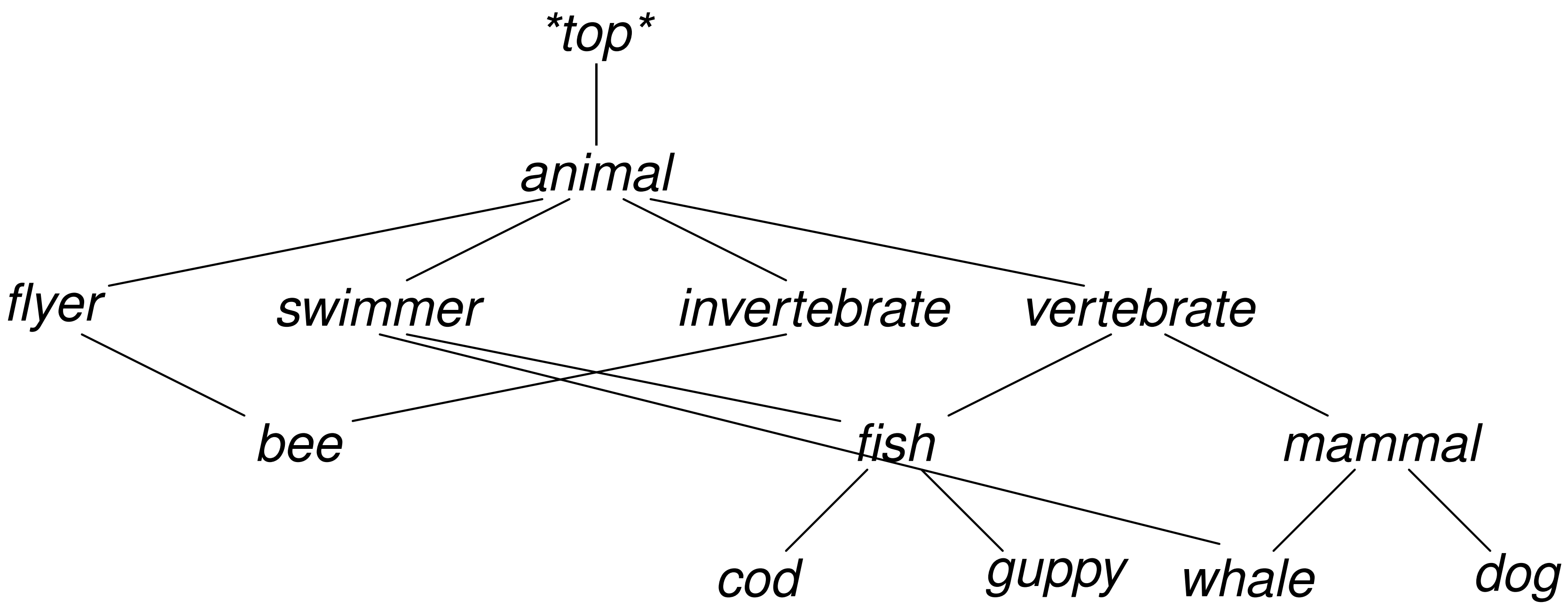
Multiple Inheritance

- *flyer* and *swimmer* 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.



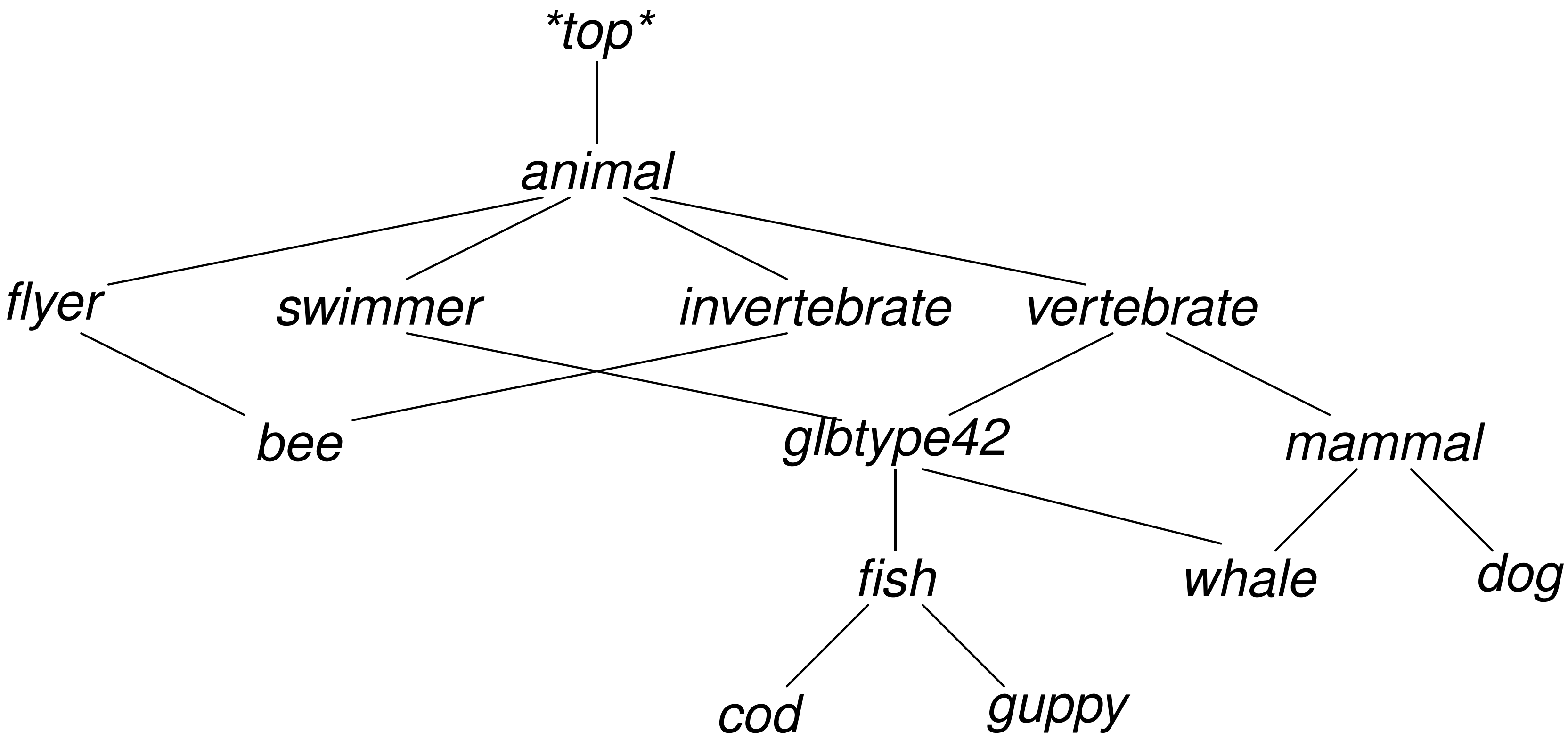
An Invalid Type Hierarchy

- *swimmer* and *vertebrate* have two joint descendants: *fish* and *whale*;
- *fish* and *whale* are incomparable in the hierarchy: glb condition violated.



Fixing the Type Hierarchy

- LKB system introduces glb types as required: ‘*swimmer-vertebrate*’.



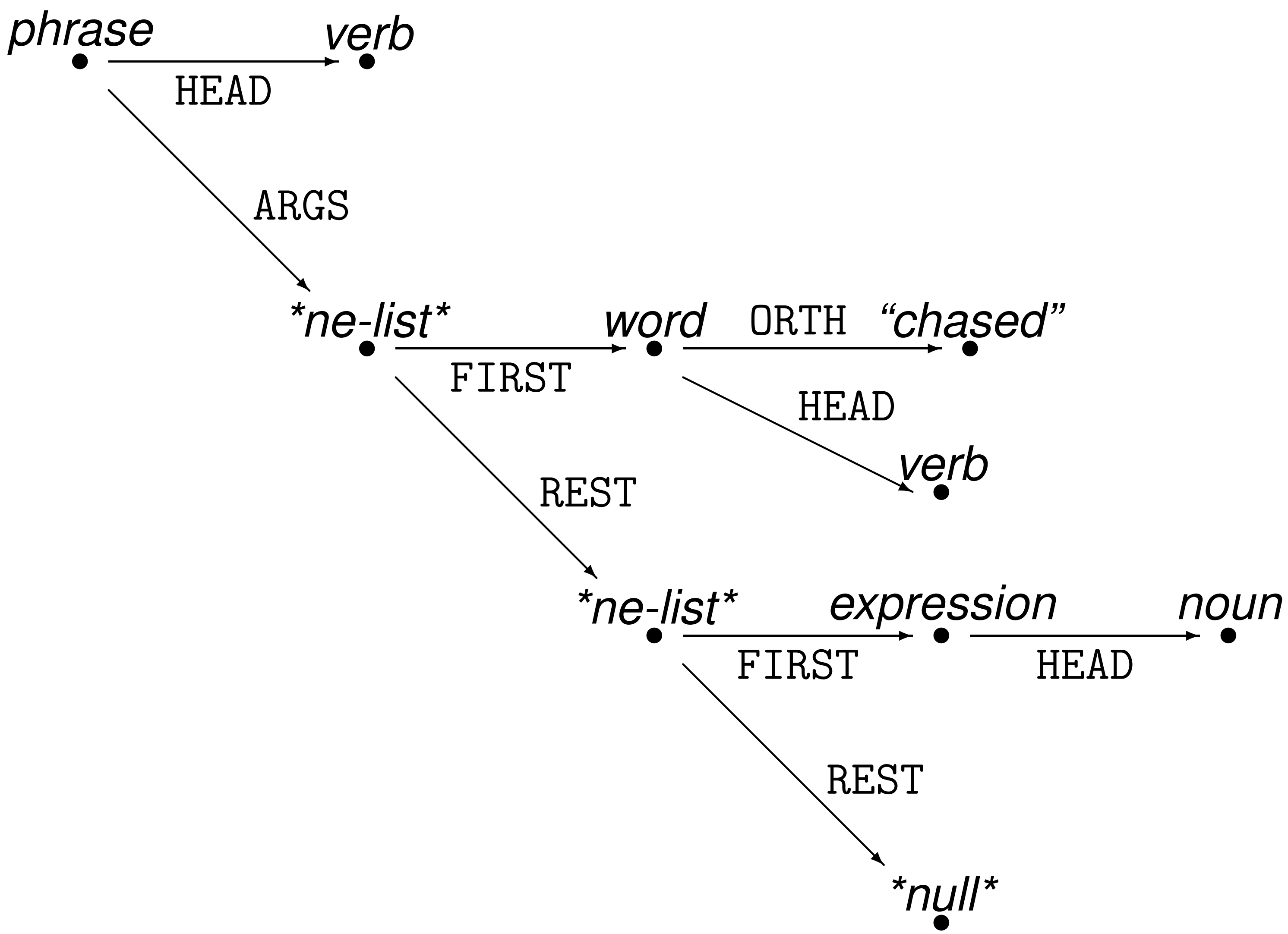
Properties of Typed Feature Structures

- **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.

Typed Feature Structure Example (as AVM)

$$
\textit{phrase}\begin{bmatrix}
\texttt{HEAD} & \textit{verb} \\
\texttt{ARGS} & \textit{*ne-list*}\begin{bmatrix}
\texttt{FIRST} & \textit{word}\begin{bmatrix}\texttt{ORTH} & \textit{“chased”} \\ \texttt{HEAD} & \textit{verb}\end{bmatrix} \\
\texttt{REST} & \textit{*ne-list*}\begin{bmatrix}
\texttt{FIRST} & \textit{expression}\begin{bmatrix}\texttt{HEAD} & \textit{noun}\end{bmatrix} \\
\texttt{REST} & \textit{*null*}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
$$

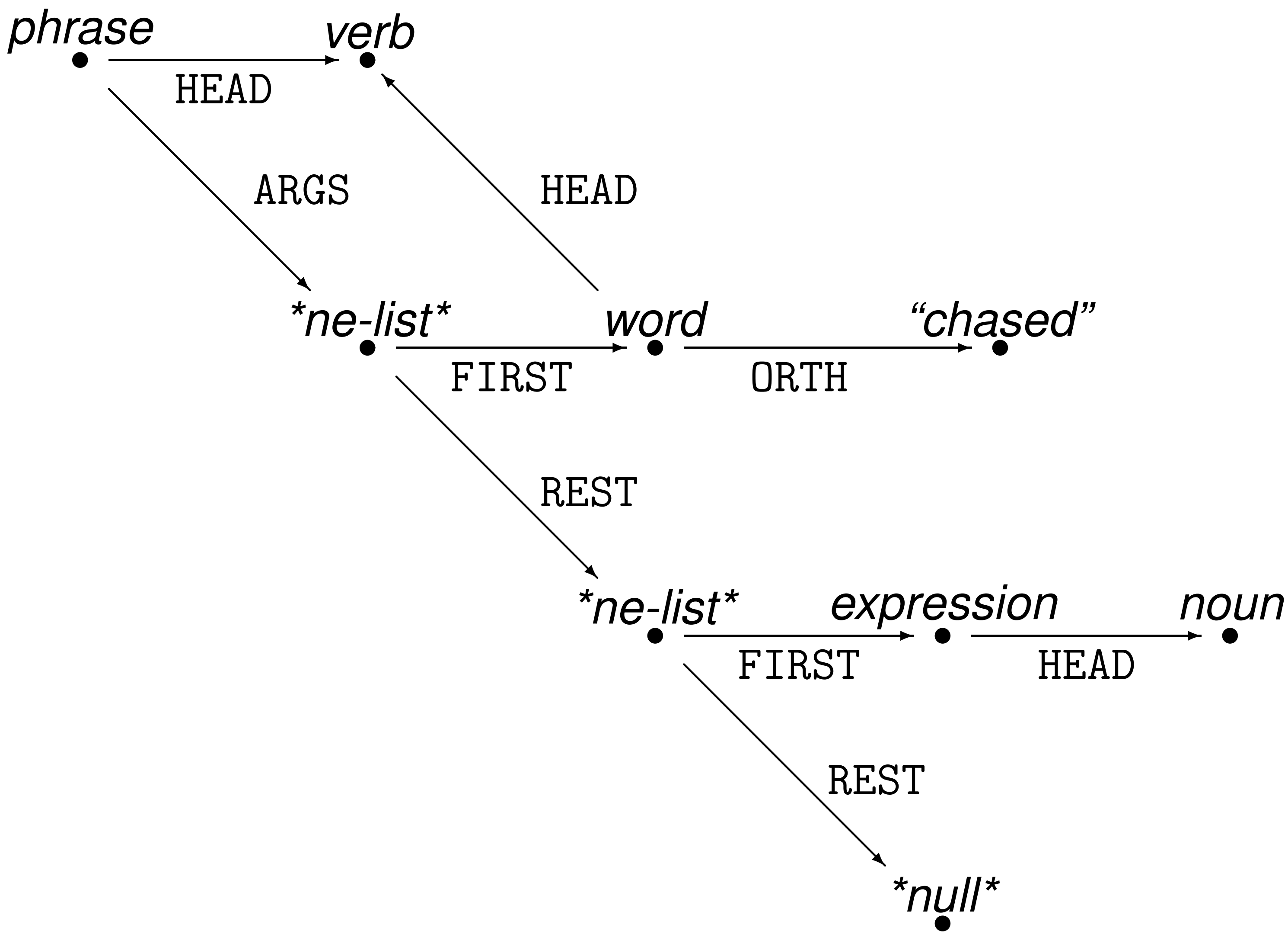
Typed Feature Structure Example (as Graph)



Typed Feature Structure Example (in TDL)

```
vp := phrase &
[ HEAD verb,
  ARGS *ne-list* &
       [ FIRST word &
               [ ORTH "chased",
                 HEAD verb ],
         REST *ne-list* &
              [ FIRST expression &
                      [ HEAD noun ],
                REST *null* ]]] .
```

Reentrancy in a Typed Feature Structure (Graph)



Reentrancy in a Typed Feature Structure (AVM)

```
phrase [ HEAD  [1] verb
         ARGS  *ne-list* [ FIRST  word [ ORTH  "chased"
                                         HEAD  [1] ]
                           REST   *ne-list* [ FIRST  expression [ HEAD  noun ]
                                              REST   *null* ] ] ]
```

Reentrancy in a Typed Feature Structure (TDL)

```
vp := phrase &
[ HEAD #head & verb,
  ARGS *ne-list* &
       [ FIRST word &
               [ ORTH "chased",
                 HEAD #head ],
         REST *ne-list* &
              [ FIRST expression &
                      [ 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;
- $_{\textit{*top*}}\begin{bmatrix}\end{bmatrix}$ 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 Subsumption: Examples

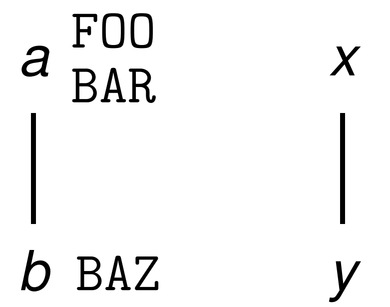
TFS$_1$: $a\begin{bmatrix}\texttt{FOO} & x \\ \texttt{BAR} & x\end{bmatrix}$

TFS$_2$: $a\begin{bmatrix}\texttt{FOO} & x \\ \texttt{BAR} & y\end{bmatrix}$

TFS$_3$: $b\begin{bmatrix}\texttt{FOO} & y \\ \texttt{BAR} & x \\ \texttt{BAZ} & x\end{bmatrix}$

TFS$_4$: $a\begin{bmatrix}\texttt{FOO} & \boxed{1}\, x \\ \texttt{BAR} & \boxed{1}\end{bmatrix}$

Signature



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 $\bot$);
- 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

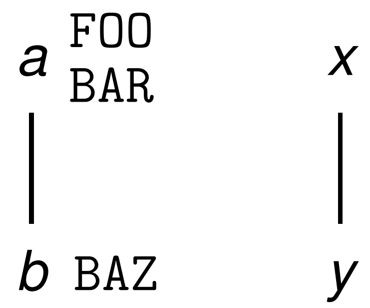
TFS$_1$: $\left._{a}\begin{bmatrix} \texttt{FOO}\ x \\ \texttt{BAR}\ x \end{bmatrix}\right.$

TFS$_2$: $\left._{a}\begin{bmatrix} \texttt{FOO}\ x \\ \texttt{BAR}\ y \end{bmatrix}\right.$

TFS$_3$: $\left._{b}\begin{bmatrix} \texttt{FOO}\ y \\ \texttt{BAR}\ x \\ \texttt{BAZ}\ x \end{bmatrix}\right.$

TFS$_4$: $\left._{a}\begin{bmatrix} \texttt{FOO}\ \boxed{1}\, x \\ \texttt{BAR}\ \boxed{1} \end{bmatrix}\right.$

Signature



TFS$_1 \sqcap$ TFS$_2 \equiv$ TFS$_2$

TFS$_1 \sqcap$ TFS$_3 \equiv$ TFS$_3$

TFS$_3 \sqcap$ TFS$_4 \equiv$ $\left._{b}\begin{bmatrix} \texttt{FOO}\ \boxed{1}\, y \\ \texttt{BAR}\ \boxed{1} \\ \texttt{BAZ}\ x \end{bmatrix}\right.$

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.

type	constraint	appropriate features
ne-list	**ne-list** [FIRST **top**, REST **list**]	FIRST and REST

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:

$$\textit{*top*}\begin{bmatrix} \texttt{HEAD} & \textit{pos} \\ \texttt{ARGS} & \textit{*list*} \end{bmatrix} \longrightarrow \textit{phrase}\begin{bmatrix} \texttt{HEAD} & \textit{pos} \\ \texttt{ARGS} & \textit{*list*} \end{bmatrix}$$

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

$$\textit{phrase}\begin{bmatrix} \texttt{HEAD} & \textit{pos} \\ \texttt{ARGS} & \textit{*list*} \end{bmatrix} \longrightarrow \textit{phrase}\begin{bmatrix} \texttt{HEAD} & \textit{pos} \\ \texttt{ARGS} & \textit{*list*} \\ \texttt{SPR} & \textit{*list*} \\ \texttt{COMPS} & \textit{*list*} \end{bmatrix}$$

More Interesting Well-Formed Unification

Type Constraints Associated to Earlier *animal* Hierarchy

$$\textit{swimmer} \rightarrow {}_{\textit{swimmer}}\begin{bmatrix}\texttt{FINS}\ \textit{bool}\end{bmatrix} \qquad \textit{mammal} \rightarrow {}_{\textit{mammal}}\begin{bmatrix}\texttt{FRIENDLY}\ \textit{bool}\end{bmatrix}$$

$$\textit{whale} \rightarrow {}_{\textit{whale}}\begin{bmatrix}\texttt{BALEEN} & \textit{bool} \\ \texttt{FINS} & \textit{true} \\ \texttt{FRIENDLY} & \textit{bool}\end{bmatrix}$$

$${}_{\textit{mammal}}\begin{bmatrix}\texttt{FRIENDLY}\ \textit{true}\end{bmatrix} \sqcap {}_{\textit{swimmer}}\begin{bmatrix}\texttt{FINS}\ \textit{bool}\end{bmatrix} \equiv {}_{\textit{whale}}\begin{bmatrix}\texttt{BALEEN} & \textit{bool} \\ \texttt{FINS} & \textit{true} \\ \texttt{FRIENDLY} & \textit{true}\end{bmatrix}$$

$${}_{\textit{mammal}}\begin{bmatrix}\texttt{FRIENDLY}\ \textit{true}\end{bmatrix} \sqcap {}_{\textit{swimmer}}\begin{bmatrix}\texttt{FINS}\ \textit{false}\end{bmatrix} \equiv \bot$$

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 expression, REST *s-list* ].
*s-null* := *null* & *s-list*.
```

Notational Conventions

- lists not available as built-in data type; abbreviatory notation in TDL:

```
< a, b > ≡ [ FIRST a, REST [ FIRST b, REST *null* ] ]
```

- underspecified (variable-length) list:

```
< a, ... > ≡ [ FIRST a, REST *list* ]
```

- difference (open-ended) lists; allow concatenation by unification:

```
<! a !> ≡ [ LIST [ FIRST a, REST #tail ], LAST #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.