Unification Parsing
Typed Feature Structures
demo: *agree grammar engineering*

Ling 571: Deep Processing Techniques for NLP
February 4, 2015

Glenn Slayden

Parsing in the abstract

- Rule-based parsers can be defined in terms of two operations:
  - Satisfiability: does a rule apply?
  - Combination: what is the result (product) of the rule?

CFG parsing

- Example CFG rule:

$$S \rightarrow NP\ VP$$

- Satisfiability:
  - Exact match of the entities on the right side of the rule
  - Do we have an NP? Do we have a VP?
  - No → try another rule. Yes →
- Combination:
  - The result of the rule application is:

$$S$$

Abstract parser desiderata

- Let’s consider a parsing formalism where the satisfiability and combination functions are combined into one operation:
- Such an operation “$\sqcup$” would:
  1. operate on two (or more) input structures
  2. produce exactly one new output structure, or
  3. sometimes fail (to produce an output structure)

  – other requirements…?

Problems with exact match

- In a CFG, this would be akin to having the "output" of a rule be its entire instance:

$$DP \rightarrow Det\ NP$$

Result: (?)

$$DP\#\#Det\#NP$$

- The problem is that this result is probably not an input (RHS) to another rule
- In fact, bottom up parsing likely would not make it past the terminals

Abstract parser desiderata

- Therefore, an additional criteria is that the putative operation "⊔"
   4. tolerate inputs which have already been specified
- This suggests that operation "⊔":
  - is information-preserving
  - monotonically incorporates ***specific*** information (from runtime inputs)
  - …into ***more general*** structures (authored rules)

Constraint-based parsing

- From graph-theory and Prolog we know that an ideal "$\sqcup$" is *graph unification*.
- The unification of two graphs is the most *specific* graph that preserves all of the *information* contained in both graphs, if such a graph is *possible*.
- We will need to define:
  - how linguistic information is represented in the graphs
  - whether two pieces of information are "compatible"
  - If compatible, which is "more specific"

Head-Driven Phrase Structure Grammar

- “HPSG,” Pollard and Sag, 1994
- Highly consistent and powerful formalism
- Monostratal, declarative, non-derivational, lexicalist, constraint-based
- Has been studied for many different languages
- Psycholinguistic evidence

HPSG foundations: Typed Feature Structures

- Typed Feature Structures (Carpenter 1992)
- High expressive power
- Parsing complexity: exponential (to the input length)
- Tractable with efficient parsing algorithms
- Efficiency can be improved with a well designed grammar

A hierarchy of scalar types

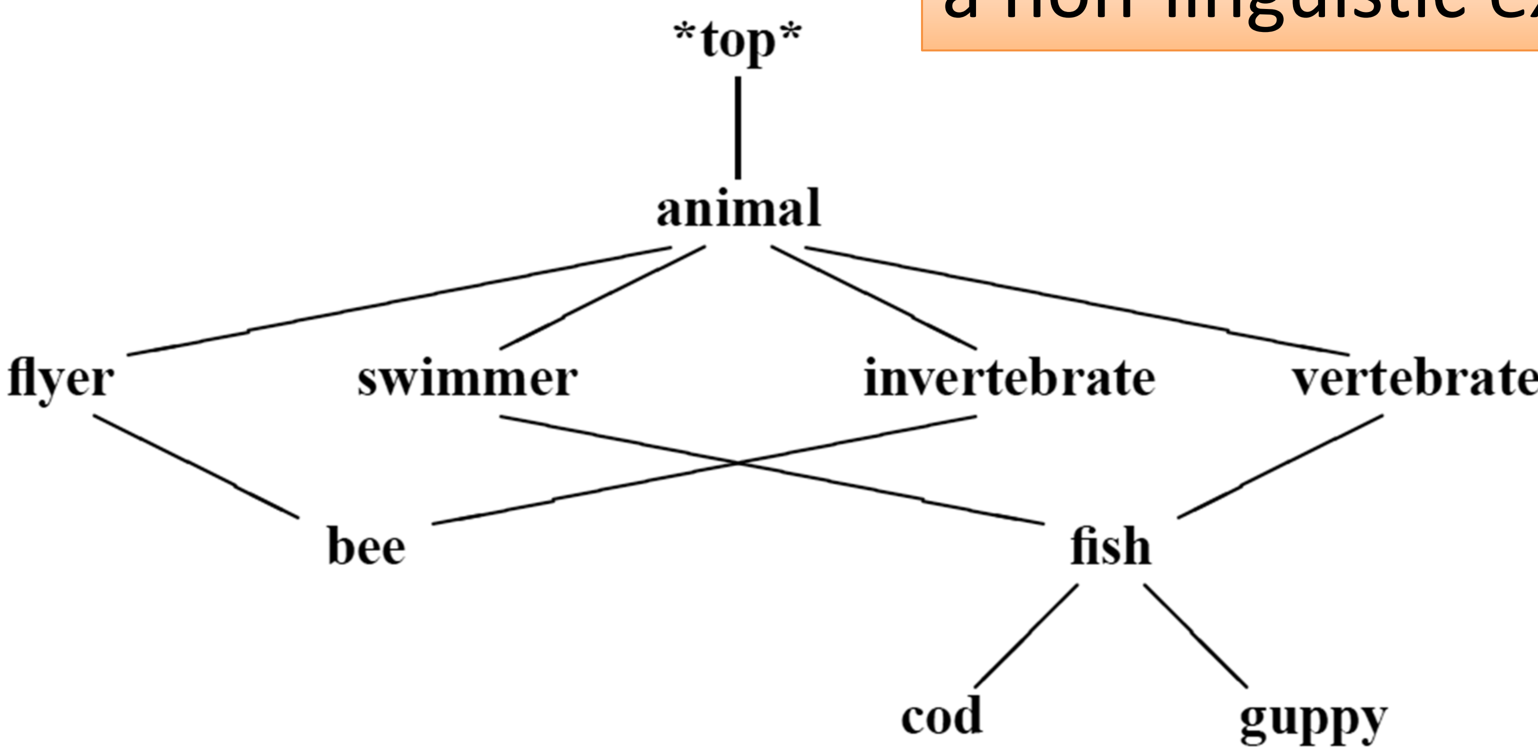
- The basis of being able constrain information is a closed universe of types
- Define a partial order of specificity over arbitrary (scalar) types
  - Type unification (vs. TFS unification)
  - A ⊔ B is defined for all types:
    - "Compatible types" A ⊔ B = C
    - "Incompatible types" A ⊔ B = ⊥

Type Hierarchy (Carpenter 1992)

- In the view of constraint-based grammar
  - A unique most general type: *top* T
  - Each non-top type has one or more parent type(s)
  - Two types are compatible *iff* they share at least one offspring type
  - Each non-top type is associated with optional constraints
    - Constraints specified in ancestor types are monotonically inherited
    - Constraints (either inherited, or newly introduced) must be compatible

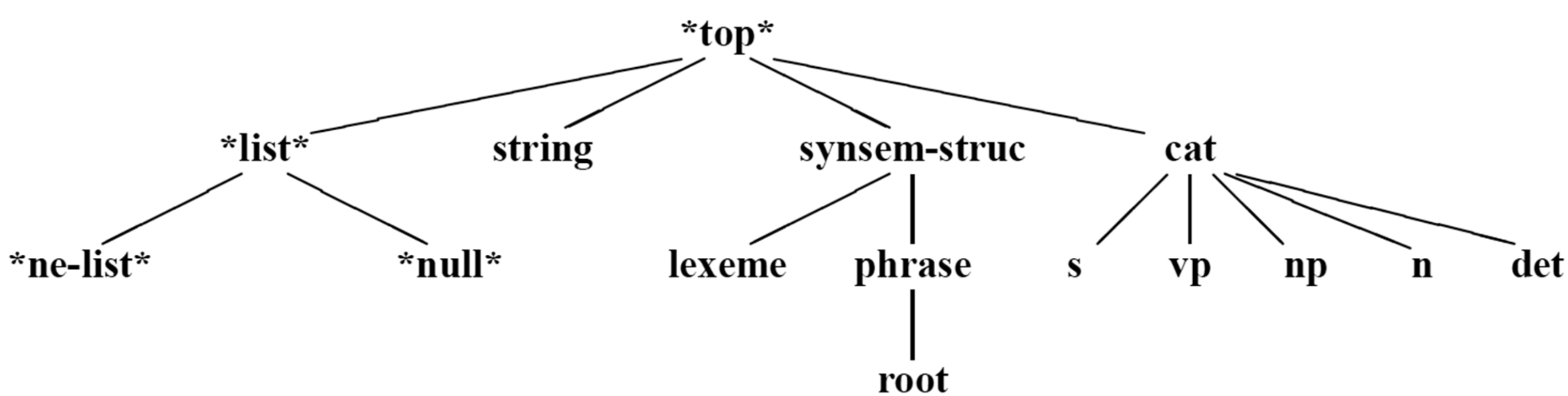
multiple inheritance

a non-linguistic example



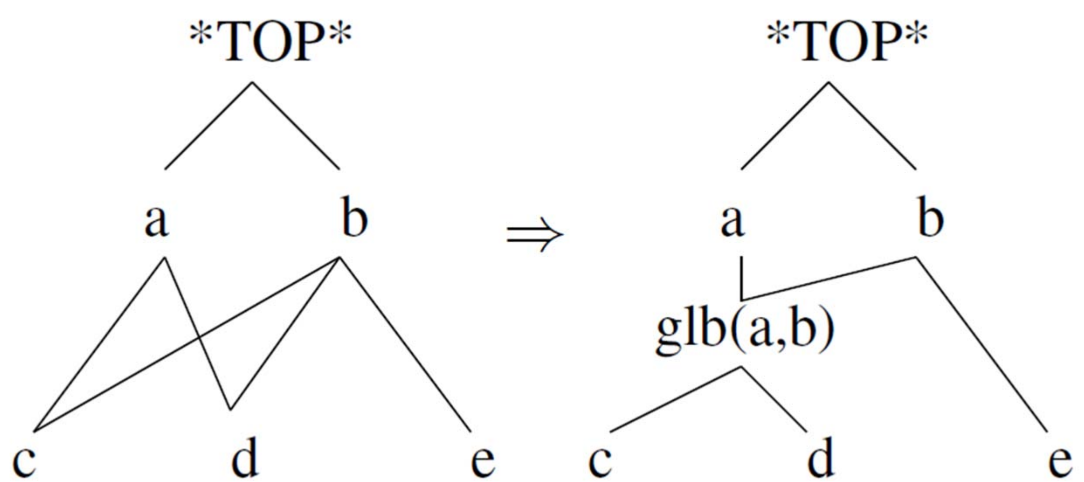
The type hierarchy

- A simple example



GLB (Greatest Lower Bound) Types

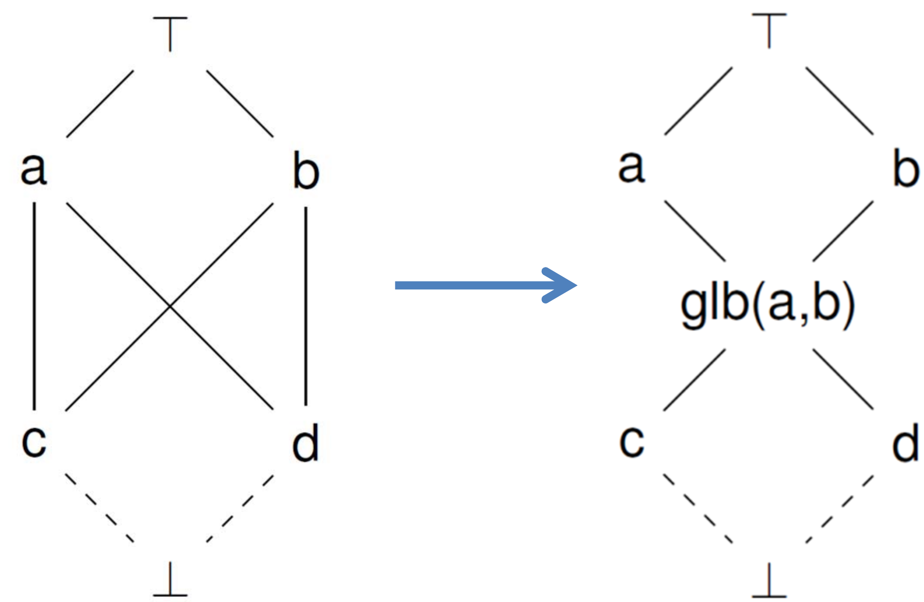
- With multiple inheritance, two types can have more than one shared subtype that neither is more general than the others
- Non-deterministic unification results
- Type hierarchy can be automatically modified to avoid this



Deterministic type unification

- Compute “bounded complete partial order” (BCPO) of the type graph

Automatically introduce GLB types so that any two types that unify have exactly one greater lowest bound

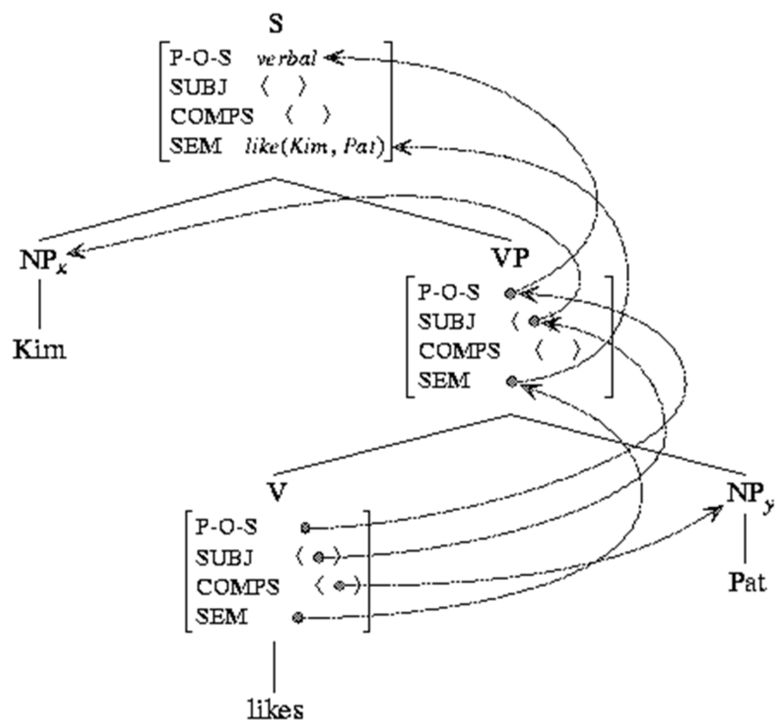


Typed Feature Structures

- [Carpenter 1992]
- High expressive power
- Parsing complexity: exponential in input length
  - Tractable with efficient parsing algorithms
  - Efficiency can be improved with a well-designed grammar

Feature Structure Grammars

- HPSG (Pollard & Sag 1994)
- http://hpsg.stanford.edu/index.html



Feature Structures In Unification-Based Grammar Development

- A feature structure is a set of attribute-value pairs
  - Or, “Attribute-Value Matrix” (AVM)
  - Each attribute (or feature) is an atomic symbol
  - The value of each attribute can be either atomic, or complex (a feature structure, a list, or a set)

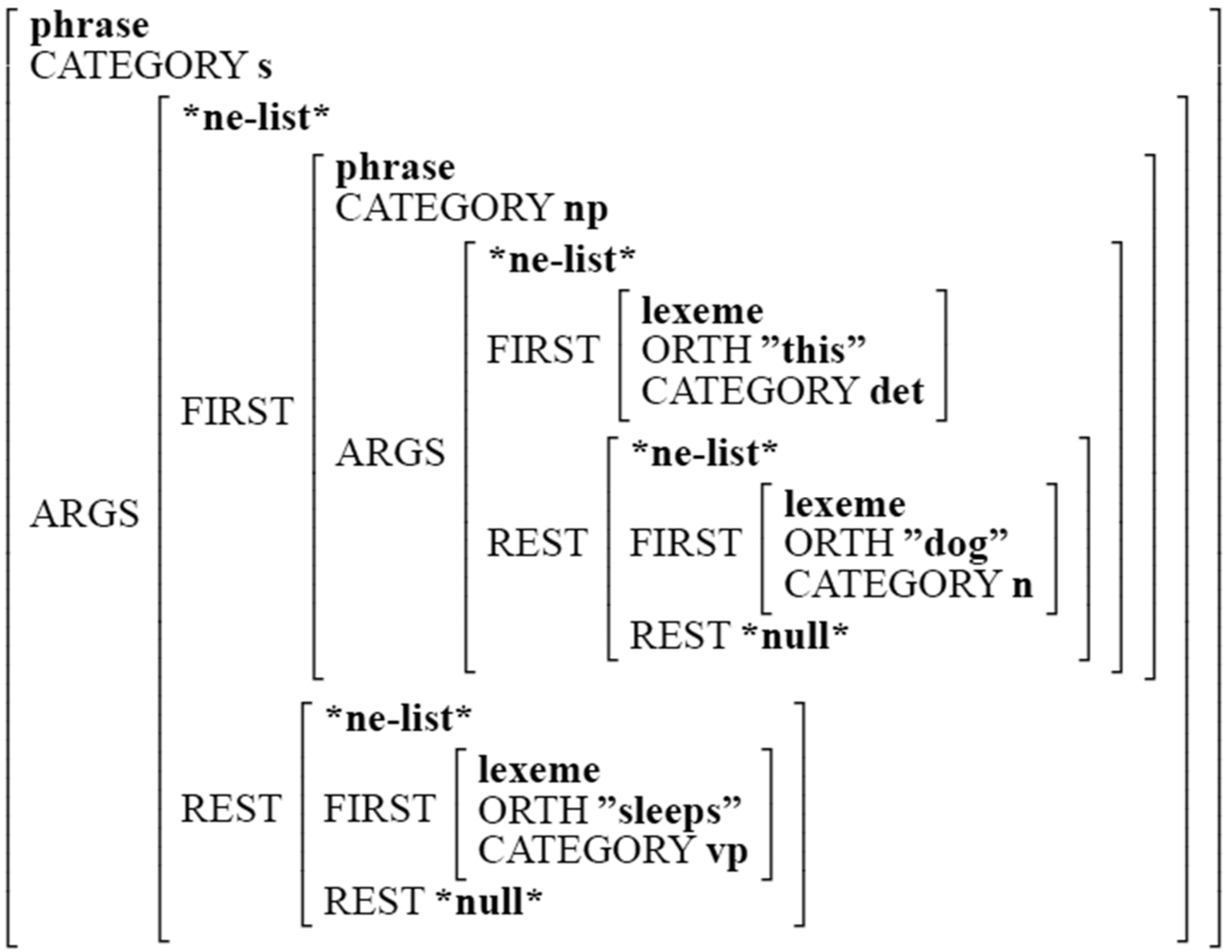
$$\begin{bmatrix} \text{CATEGORY} & \textit{noun-phrase} \\ \text{AGREEMENT} & \begin{bmatrix} \text{PERSON} & \textit{3rd} \\ \text{NUMBER} & \textit{sing} \end{bmatrix} \end{bmatrix}$$

Typed Feature Structure

- A typed feature structure is composed of two parts
  - A **type** (from the scalar type hierarchy)
  - A (possibly empty) set of attribute-value pairs ("**Feature Structure**") with each value being a TFS

$$\begin{bmatrix} \text{CATEGORY} & \textit{noun-phrase} \\ \text{AGREEMENT} & \begin{bmatrix} \text{PERSON} & \textit{3rd} \\ \text{NUMBER} & \textit{sing} \end{bmatrix} \end{bmatrix}$$

Typed Feature Structure (TFS)



Properties of TFSes

- Finiteness

  a typed feature structure has a finite number of nodes
- Unique root and connectedness

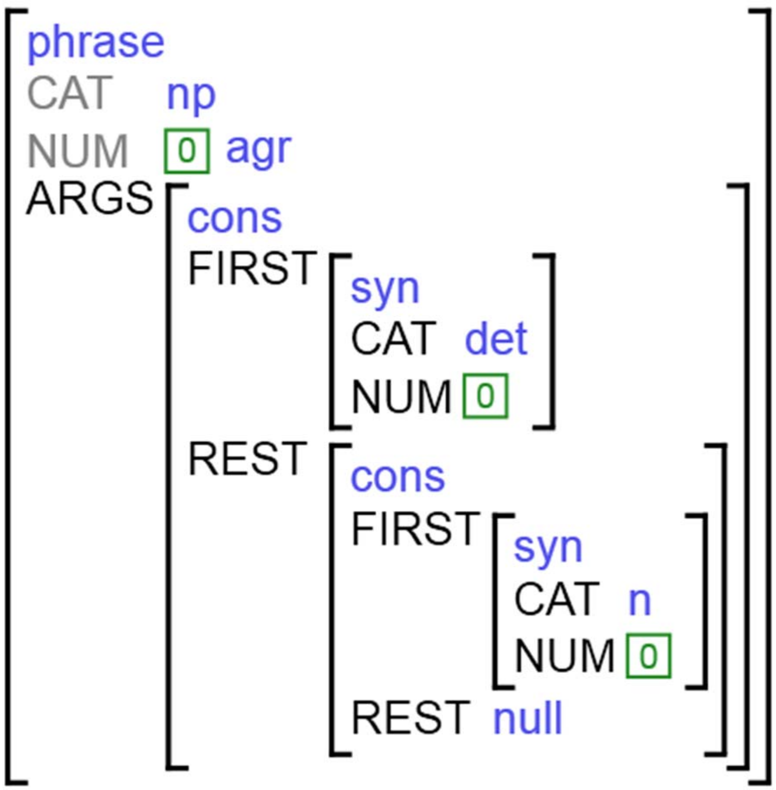
  a typed feature structure has a unique root node; apart from the root, all nodes have at least one parent
- No cycles

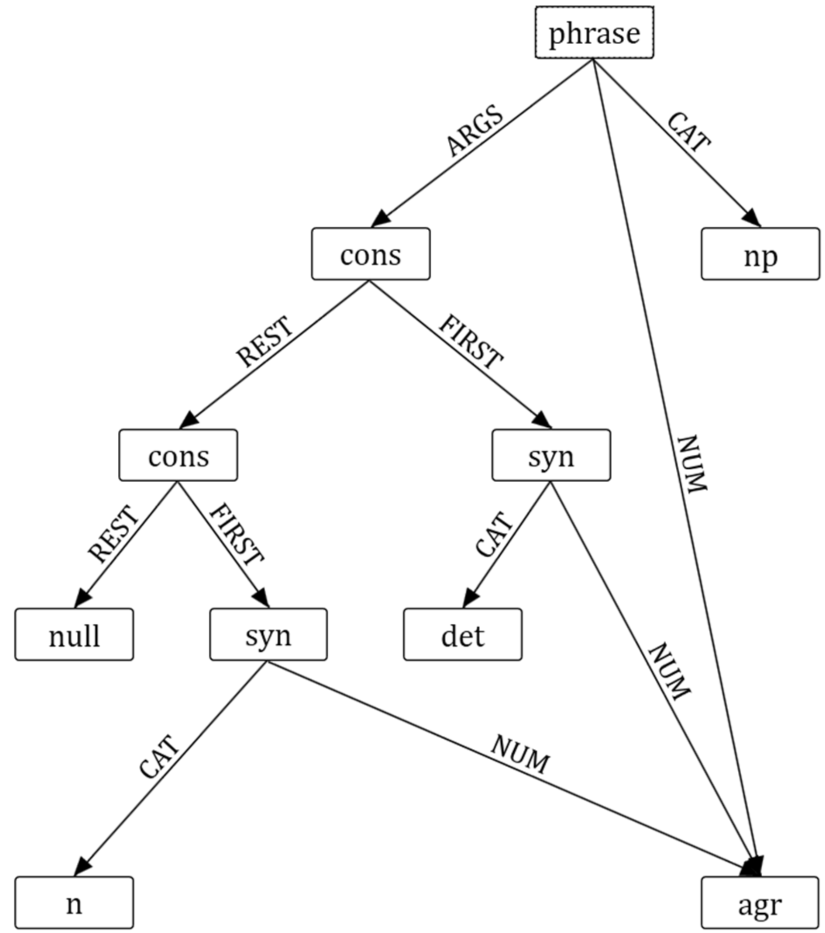
  no node has an arc that points back to the root node or to another node that intervenes between the node itself and the root
- Unique features

  no node has two features with the same name and different values
- Typing

  each node has single type which is defined in the hierarchy

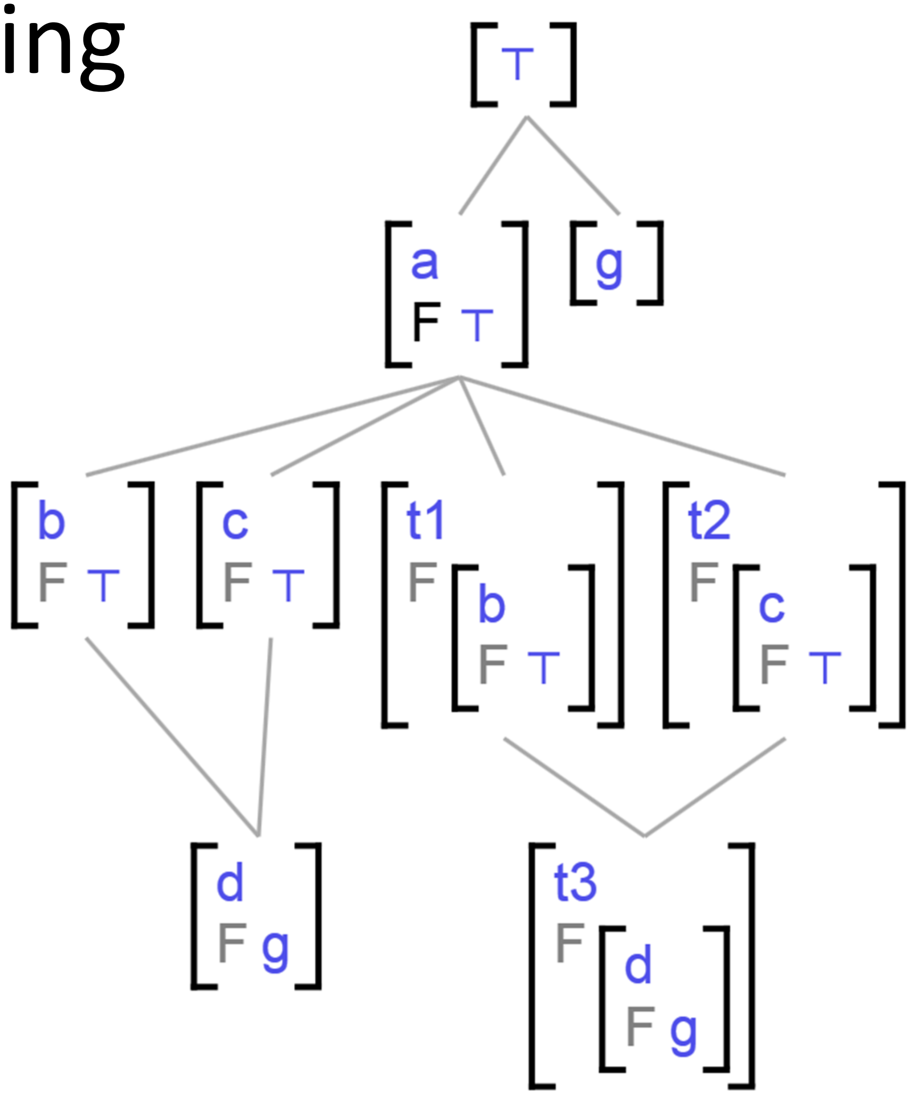
TFS equivalent views





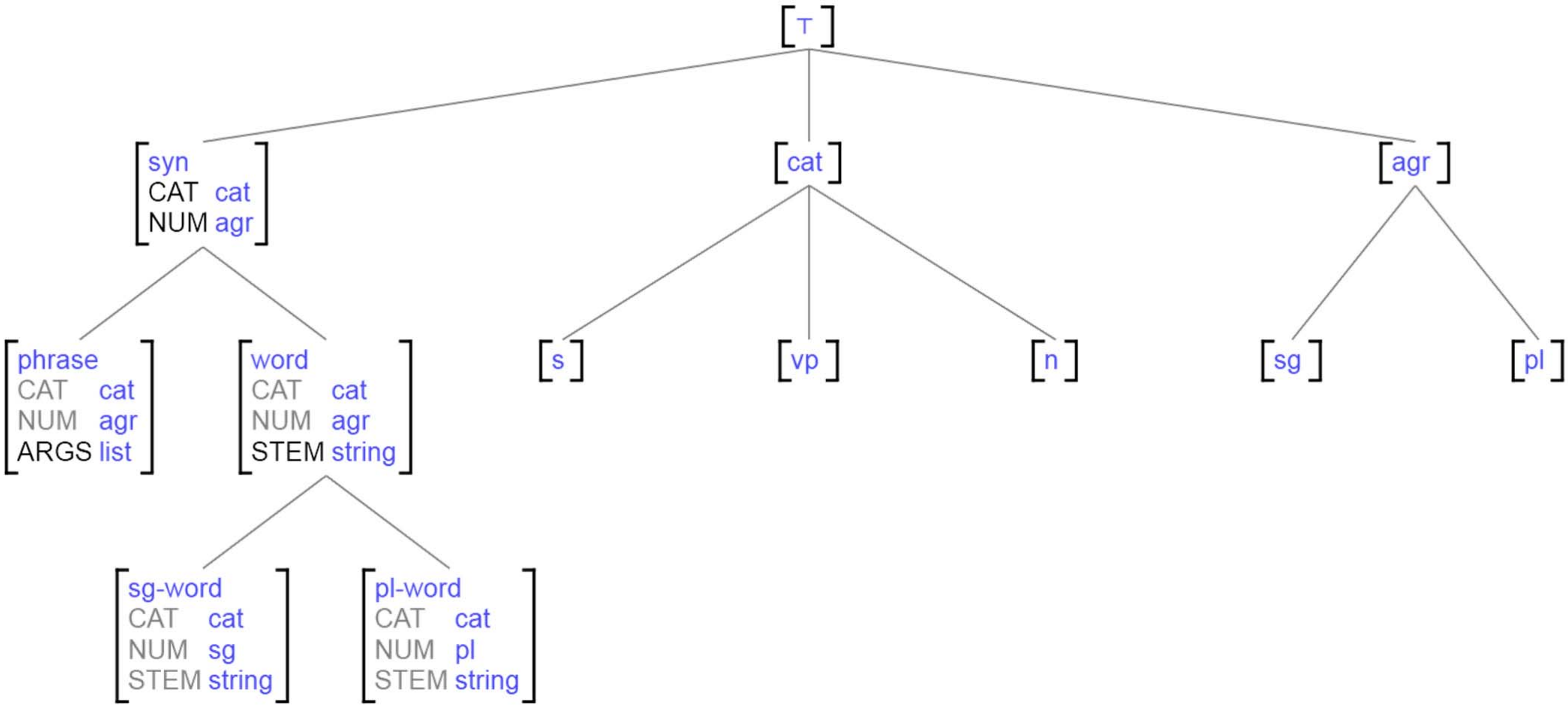
TFS partial ordering

- Just as the (scalar) type hierarchy is ordered, TFS instances can be ordered by subsumption



TFS hierarchy

- The backbone of the TFS hierarchy is the scalar type hierarchy; but note that TFS [agr] is *not* the same entity as type agr

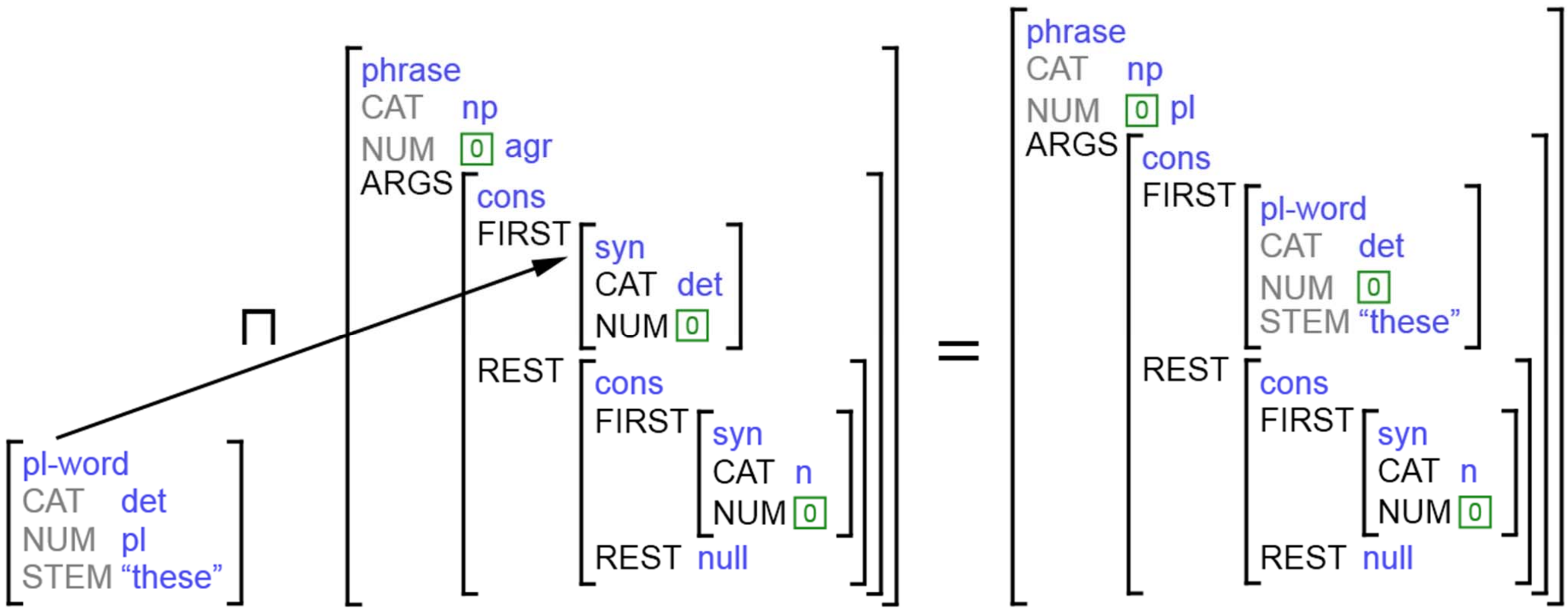


Unification

The unification result on two TFSes TFS_a and TFS_b is:

- $\bot$, if either one of the following:
  - type a and b are incompatible
  - unification of values for attribute X in TFS_a and TFS_b returns $\bot$
- a new TFS, with:
  - the most general shared subtype of a and b
  - a set of attribute-value pairs being the results of unifications on sub-TFSes of TFS_a and TFS_b

TFS Unification

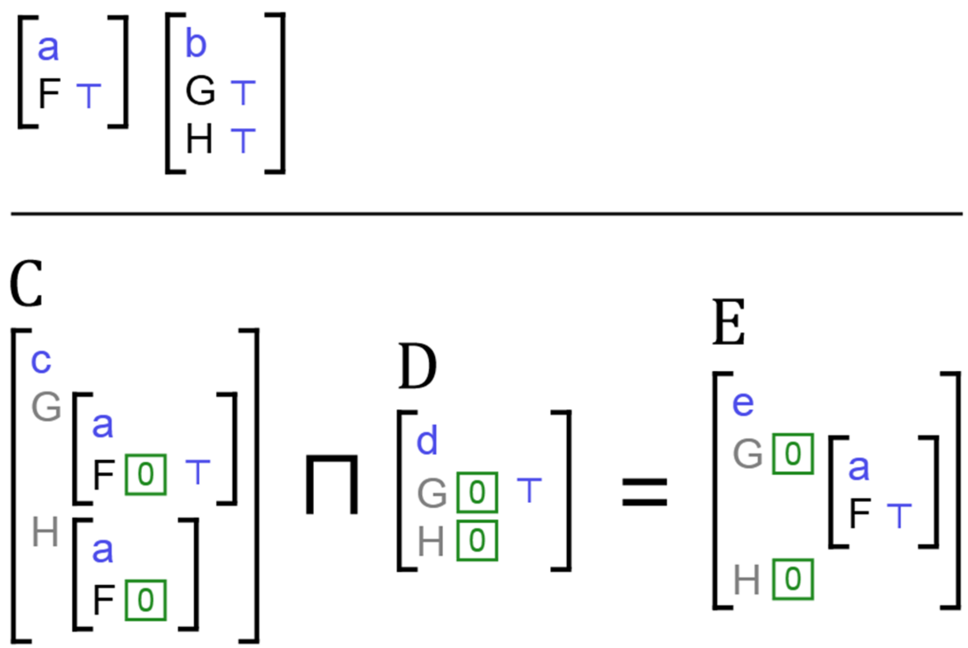


Lexical entry   Grammar rule   Partial rule application

TFS unification

TFS unification has much subtlety

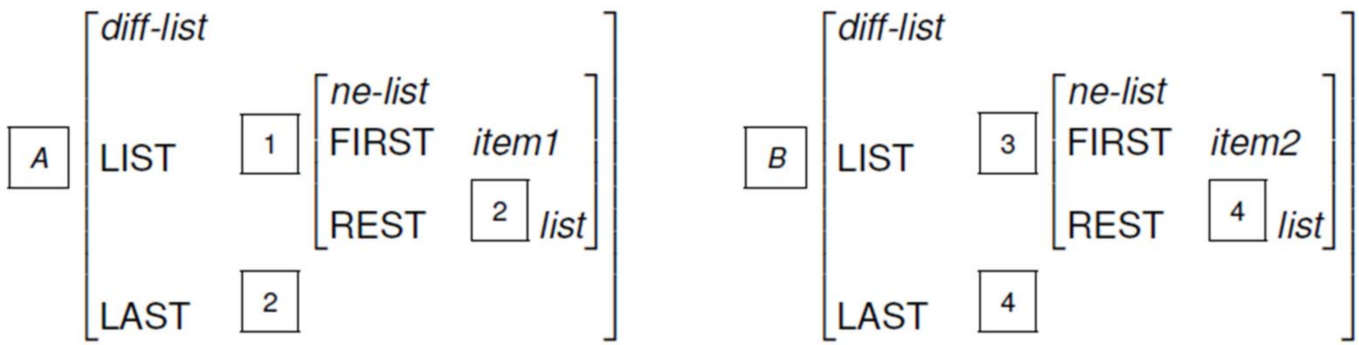
For example, it can render authored co-references vacuous



The condition on F, present in TFS C, has collapsed in E

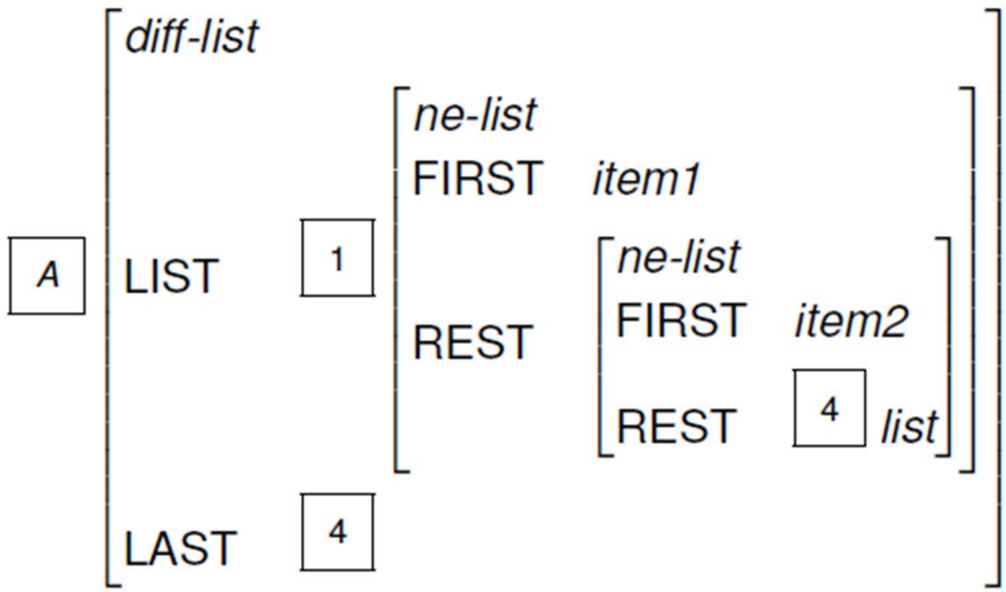
Building lists with unification

- A *difference list* embeds an open-ended list into a container structure that provides a 'pointer' to the end of the ordinary list.



- Using the LAST pointer of difference list A we can append A and B by
  - unifying the front of B (i.e. the value of its LIST feature) into the tail of A (its LAST value) and
  - using the tail of difference list B as the new tail for the result of the concatenation.

Result of appending the lists



Representing Semantics in Typed Feature Structures

Semantics desiderata

- For each sentence admitted by the grammar, we want to produce a meaning representation suitable for applying rules of inference.

"This fierce dog chased that angry cat."

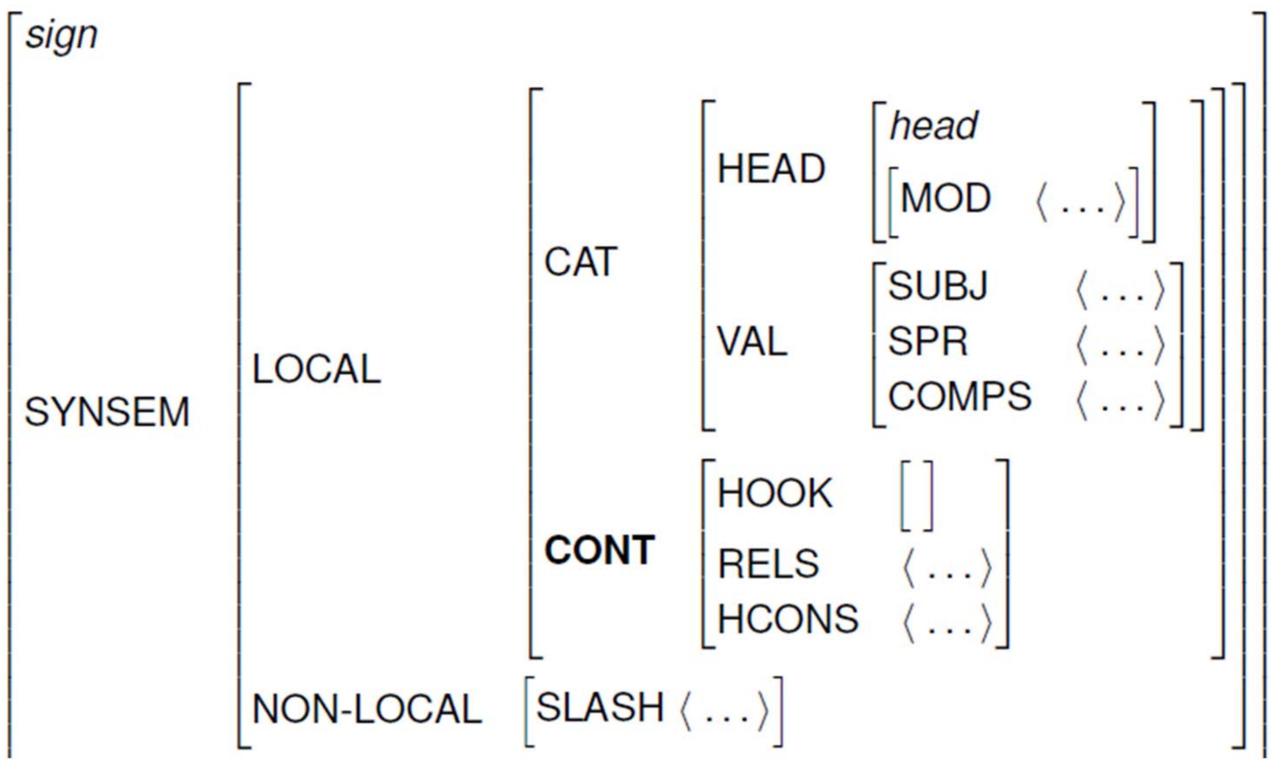
$$\text{this}(x) \land \text{fierce}(x) \land \text{dog}(x) \land$$
$$\text{chased}(e, x, y) \land$$
$$\text{that}(y) \land \text{angry}(y) \land \text{cat}(y)$$

Semantics desiderata

- Compositionality
  - The meaning of a phrase is composed of the meanings of its parts.
- Existing machinery
  - Unification is the only mechanism we use for constructing semantics in the grammar.

Semantics in feature structures

- Semantic content in the CONT attribute of every word and phrase



Semantics formalism: MRS

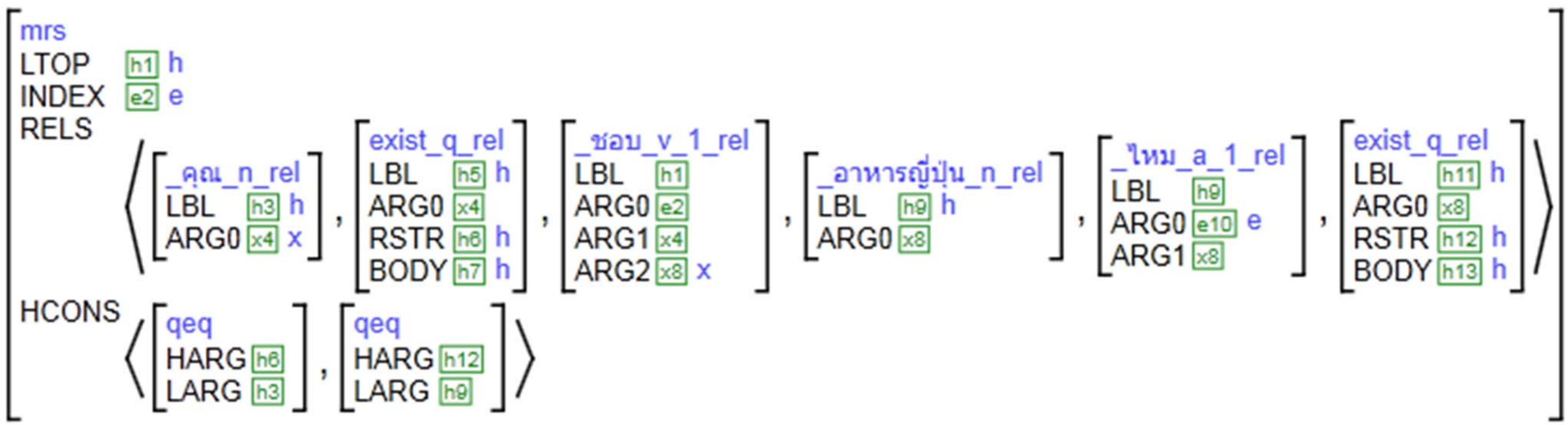
- Minimal Recursion Semantics

  Copestake, A., Flickinger, D., Pollard, C. J., and Sag, I. A. (2005). *Minimal recursion semantics: an introduction*. Research on Language and Computation, 3(4):281–332.

- Used across DELPH-IN projects
- The value of CONT for a sentence is essentially a list of relations in the attribute RELS, with the arguments in those relations appropriately linked:
  - Semantic relations are introduced by lexical entries
  - Relations are appended when words are combined with other words or phrases.

MRS: example

คุณชอบอาหารญี่ปุ่นไหม



DELPH-IN consortium

DELPH-IN Consortium

- An informal collaboration of about 20 research sites worldwide focused on deep linguistic processing since ~2002
  - DFKI Saarbrücken GmbH, Germany
  - Stanford University, USA
  - University of Oslo, Norway
  - Saarland University, Germany
  - University of Washington, Seattle, USA
  - Nanyang Tecnological University, Singapore
  - …many others
- http://www.delph-in.net

Key DELPH-IN Projects

- English Resource Grammar (ERG)

  Flickinger 2002, www.delph-in.net/erg
- The Grammar Matrix

  Bender et al. 2002, www.delph-in.new/matrix
- Other large grammars

  JACY (Japanese, Siegel and Bender 2002)
  GG; Cheetah (German; Crysmann; Cramer and Zhang 2009)
  Many others: http://moin.delph-in.net/GrammarCatalogue
- Operational instrumentation of grammars

  [incr tsdb()] (Oepen and Flickinger 1998)
- Joint-reference formalism tools

English Resource Grammar

(Flickinger 2002)

- A large, open source HPSG computational grammar of English
- 20+ years of work
- Likely the most competent general domain, rule-based grammar of any language
- Redwoods treebank

Grammar Matrix

- Rapid prototyping of computational grammars for new languages
- Also for computational typology research
- From a Web-based questionnaire, produce a customized working starter grammar

  http://www.delph-in.net/matrix/customize/

Relevant DELPH-IN research

- Morphological pre-processing
- Chart parsing optimizations
- Generation techniques
- Ambiguity packing
- Parse selection
  - maximum-entropy parse selection model

Chart parsing efficiency

- parser optimizations
  - "quick-check"
  - ambiguity packing
  - "chart dependencies" phase
  - spanning-only rules
  - rule compatibility pre-checks
  - key-driven
  - grammar design for faster parsing

Ambiguity packing

- Primary approach to combating parse intractability
- Every new feature structure is checked for a subsumption relationship with existing TFSs.
  - Subsumed TFSs are 'packed' into the more general structure
  - They are excluded from continuing parse activities
  - 'Unpacking' recovers them after the parse is complete
- *agree*: concurrent implementation of a DELPH-IN method
  - Oepen and Carroll 2000
  - Proactive/retroactive; subsumption/equivalence
- Applicable to parsing and generation

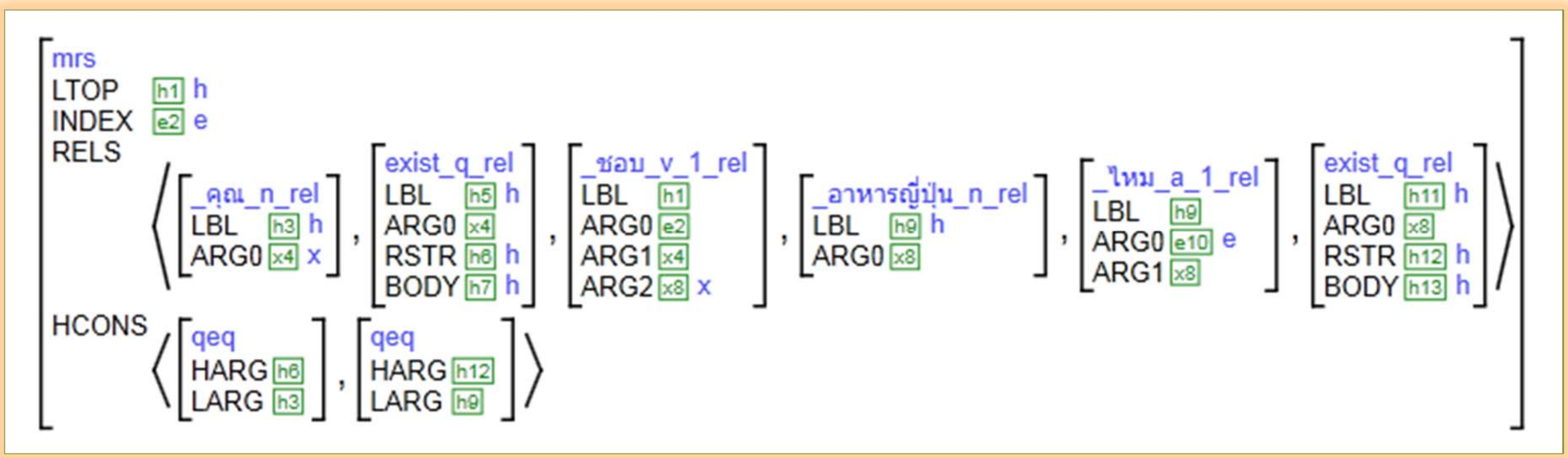
Parsing *vs.* Generation

- DELPH-IN computational grammars are bi-directional:

คุณชอบอาหารญี่ปุ่นไหม

Parsing

Generation



Generation

- Generation uses the same bottom-up chart parser…

  …with a different adjacency/proximity condition
  - Instead of joining adjacent words (parsing) the generator joins mutually-exclusive EPs
- Trigger rules
  - Required for postulating semantically vacuous lexemes
- Index accessibility filtering
  - Futile hypotheses can be intelligently avoided
- Skolemization
  - Inter-EP relationships (‘variables’) are burned-in to the input semantics to guarantee proper semantics

DELPH-IN Joint Reference Formalism

- Key focus of DELPH-IN research: computational Head-driven Phrase Structure Grammar
  HPSG, Pollard & Sag 1994
- TDL: Type Description Language
  Krieger & Schafer 1994
- A minimalistic constraint-based typed feature structure (TFS) formalism that maintains computational tractability
  Carpenter 1992
- MRS: Minimum Recursion Semantics
  Copestake et al. 1995, 2005
- Multiple toolsets: LKB, PET, Ace, agree
- Committed to open source

TDL: Type Description Language

- A text-based format for authoring constraint-based grammars

```
demonst-numcl-lex := raise-sem-lex-item &
    [ SYNSEM.LOCAL [ CAT [ HEAD numcl & [ MOD < > ],
                           VAL [ COMPS < [ OPT +, LOCAL [ CAT.HEAD num,
                                                          CONT.HOOK [ XARG #xarg,
                                                                      LTOP #larg ] ] ] >,
                                 SPEC < >,
                                 SPR < >,
                                 SUBJ < > ] ],
                     CONT.HOOK [ XARG #xarg, LTOP #larg ] ] ].
```

TDL: type definition language

```
;;; Types
string := *top*.
*list* := *top*.
*ne-list* := *list* &
[ FIRST *top*,
REST *list* ].

*null* := *list*.
synsem-struc := *top* &
    [ CATEGORY cat,
    NUMAGR agr ].

cat := *top*.
s := cat.
np := cat.
vp := cat.
det := cat.
n := cat.
agr := *top*.
sg := agr.
```

```
;;; Lexicon
this := sg-lexeme & [ ORTH "this", CATEGORY det ].
these := pl-lexeme & [ ORTH "these", CATEGORY det ].
sleep := pl-lexeme & [ ORTH "sleep", CATEGORY vp ].
sleeps := sg-lexeme & [ ORTH "sleeps", CATEGORY vp ].
dog := sg-lexeme & [ ORTH "dog", CATEGORY n ].
dogs := pl-lexeme & [ ORTH "dogs", CATEGORY n ].
```

```
;;; Rules
s_rule := phrase & [ CATEGORY s, NUMAGR #1, ARGS [ FIRST [
CATEGORY np,...
```

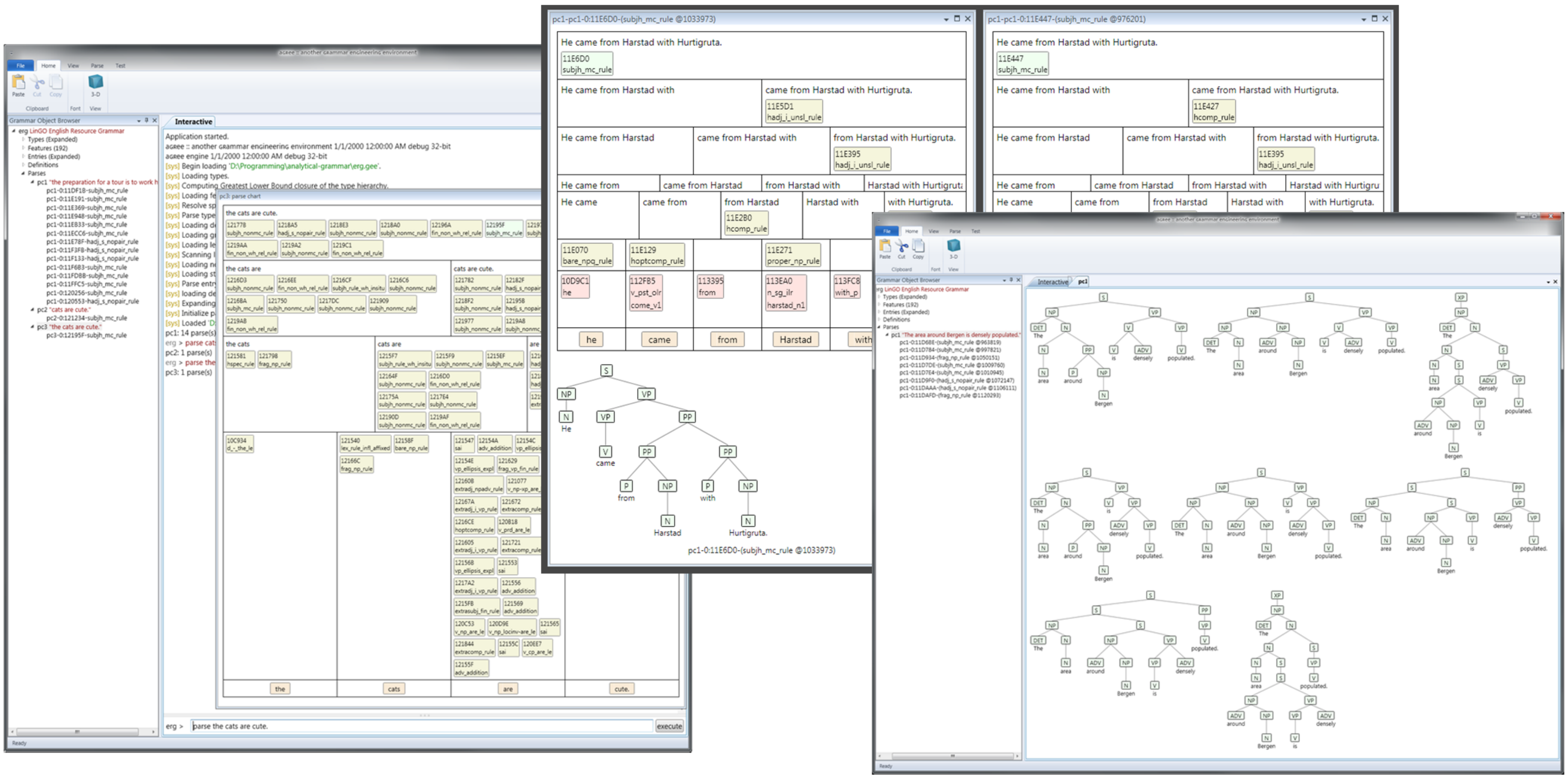
‘agree’ grammar engineering

agree grammar engineering environment

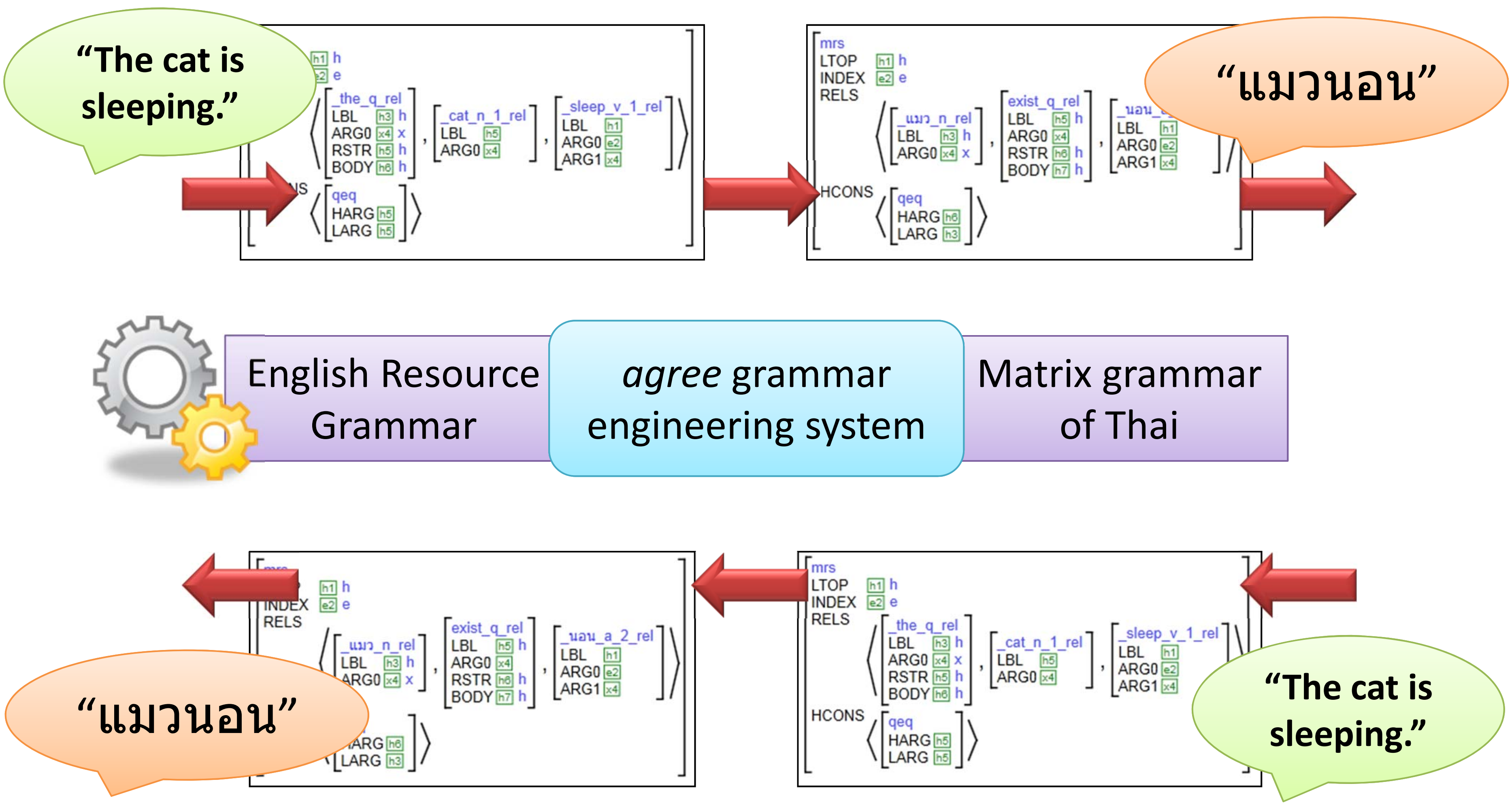
- A new toolset for the DELPH-IN formalism
  - Started in 2009
  - Joins the LKB (1993), PET (2001) and ACE (2011)
- All-new code (C#), for .NET/Mono platforms
- Concurrency-enabled from the ground-up
  - Thread-safe unification engine
  - Lock-free concurrent parse/generation chart
- Supports both parsing and generation
  - Also, DELPH-IN compatible morphology unit

agree WPF

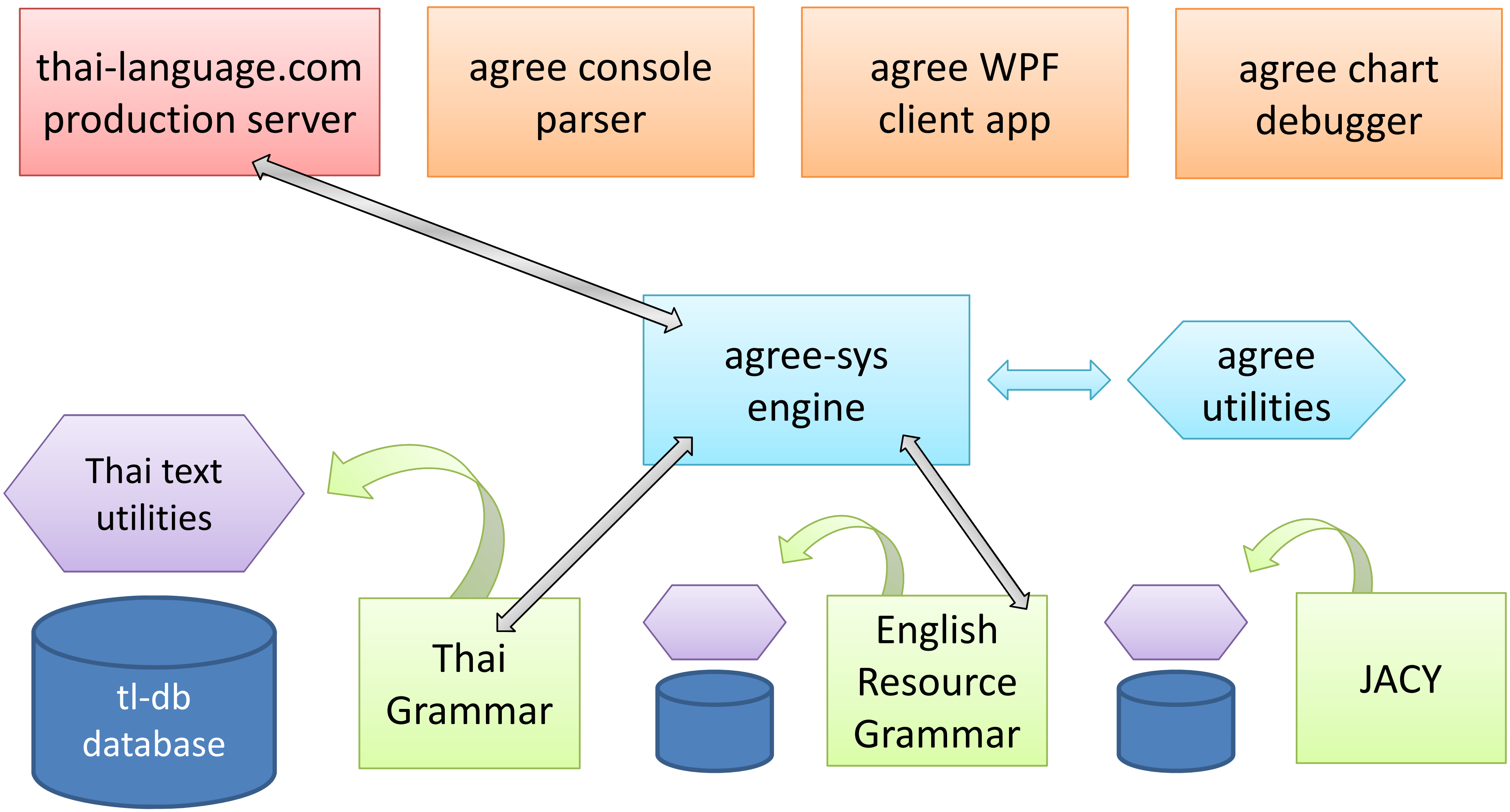
- For Windows, there is a graphical client application



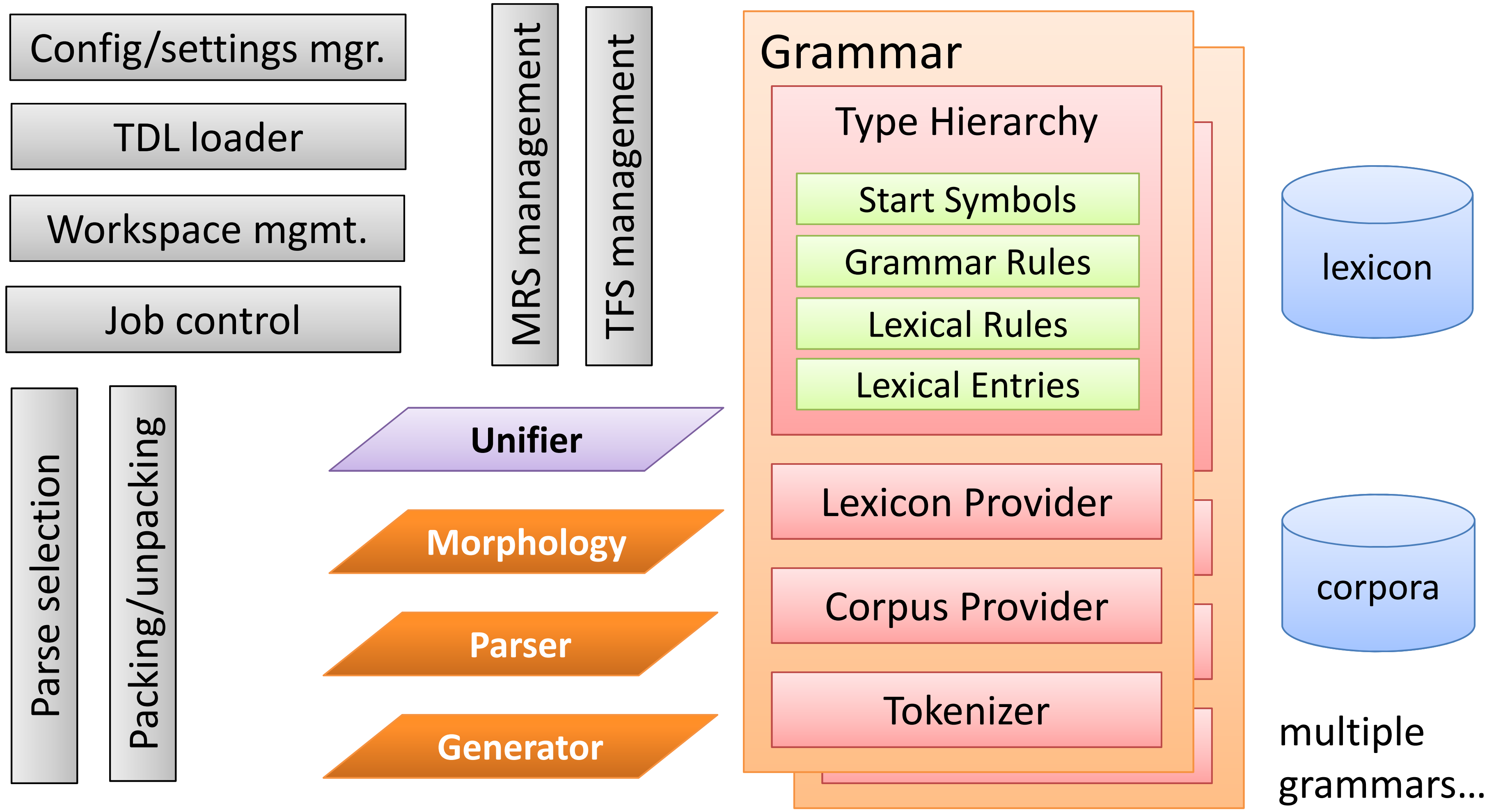
Proposed “deep” Thai-English system



Project components

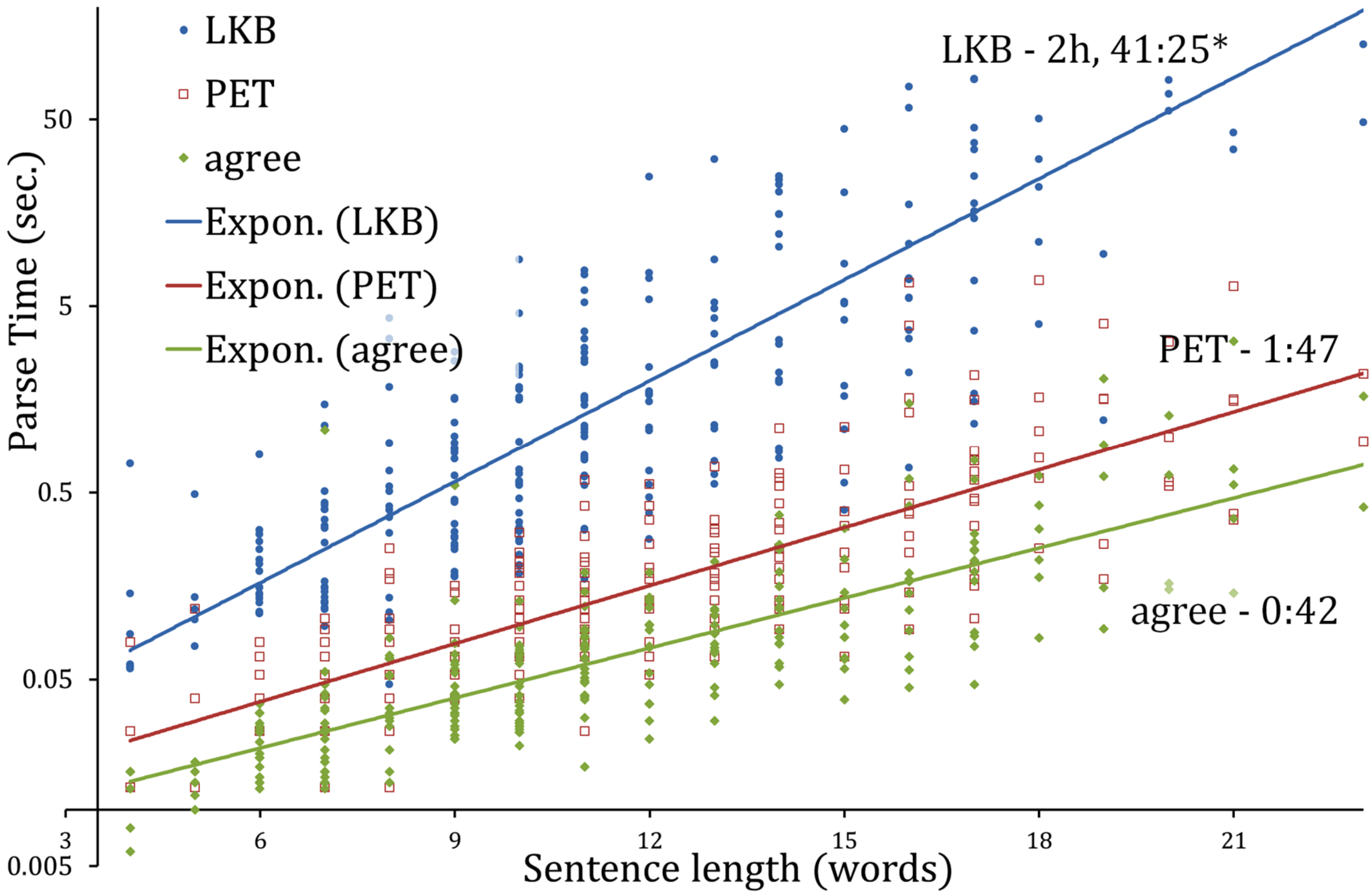


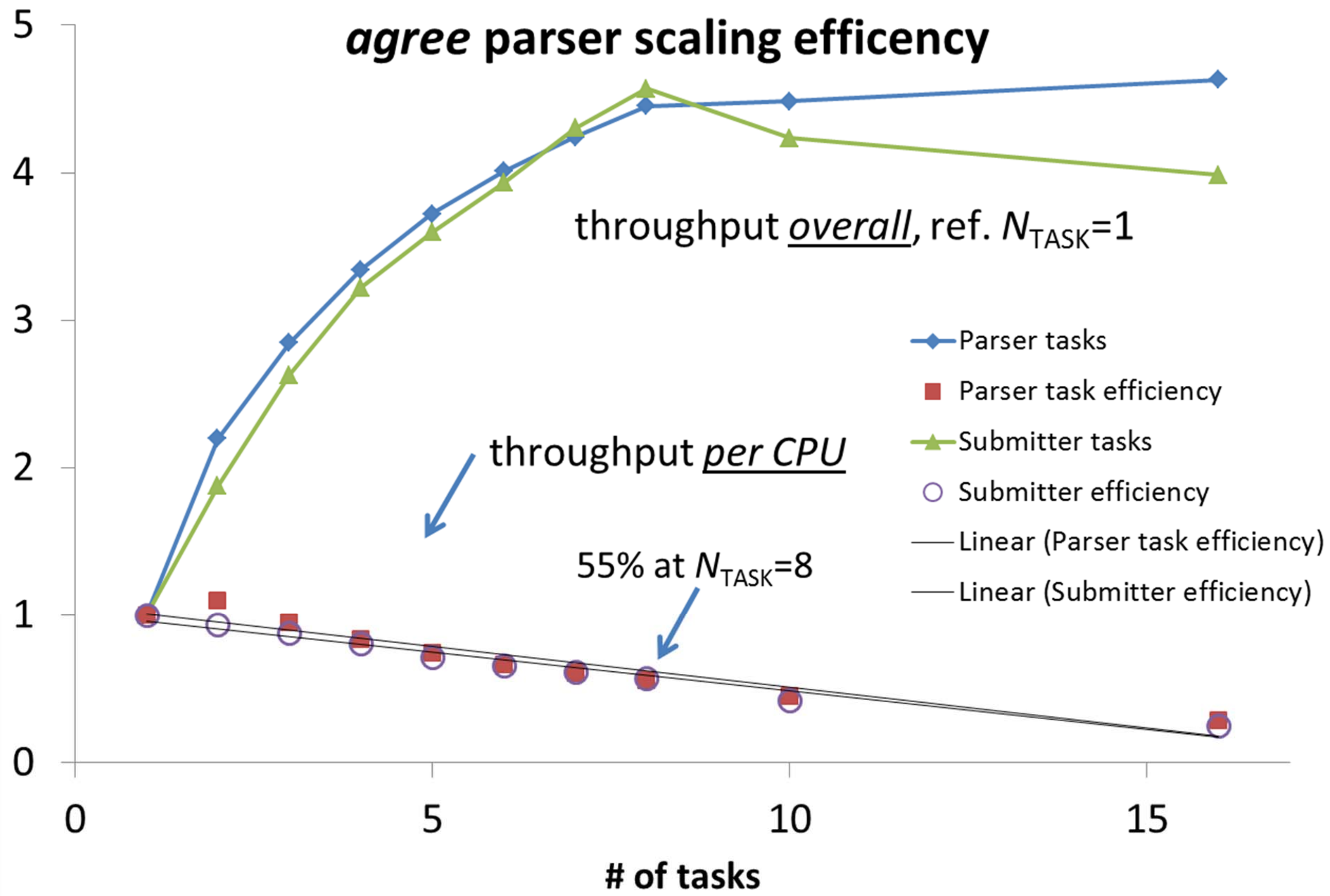
agree-sys engine components



agree parser performance

Time to parse 287 sentences from 'hike' corpus; *agree* concurrency x8




agree parser scaling efficency
throughput *overall*, ref. N_{TASK}=1
throughput *per CPU*
55% at N_{TASK}=8
Parser tasks
Parser task efficiency
Submitter tasks
Submitter efficiency
Linear (Parser task efficiency)
Linear (Submitter efficiency)
0
1
2
3
4
5
0
5
10
15
# of tasks

agree Mono

- *agree* is primarily tested and developed on Windows (.NET runtime environment)
- Mac and Linux builds have also been tested:

```
File  Edit  View  Terminal  Help
glenn@linux:~/analytical-grammar$ mono agree.exe /home/glenn/erg/erg.gee -parse "The child has the flu."
========== Loading grammar file ============================================================
loaded 52 quick-check paths
types 4821 closed 7785 glb 2964 ops 7799129
============================================================ ok: 7279 ms ===================
Regression test succeeded.
garbage report disabled on Mono
========== Parsing... ======================================================================
0 0 [The child has the flu.] 1 parses.  0.258 sec.
S (NP (DET N) VP (V NP (DET N)))
garbage report disabled on Mono
============================================================ ok: 1956 ms ===================

glenn@linux:~/analytical-grammar$
```

agree demo…