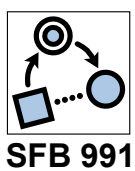
Grammar Implementation with Lexicalized Tree Adjoining Grammars and Frame Semantics

Grammar implementation with XMG

Laura Kallmeyer, Timm Lichte, Rainer Osswald & Simon Petitjean

University of Düsseldorf

DGfS Fall School, September 18, 2017



Outline

Last week

Mon: introduction to LTAG

Tue: syntactic analyses with LTAG I, derivation trees, feature structures

Wed: syntactic analyses with LTAG II, introduction to LTAG semantics

Thu: introduction to frame semantics

Fri: putting things together

This week

Mon: introduction to grammar engineering and XMG

Tue: implementing syntax with XMG

Wed: implementing semantics with XMG

Thu: parsing implemented grammars with TuLiPA

Fri: conclusion

Grammar engineering: the task

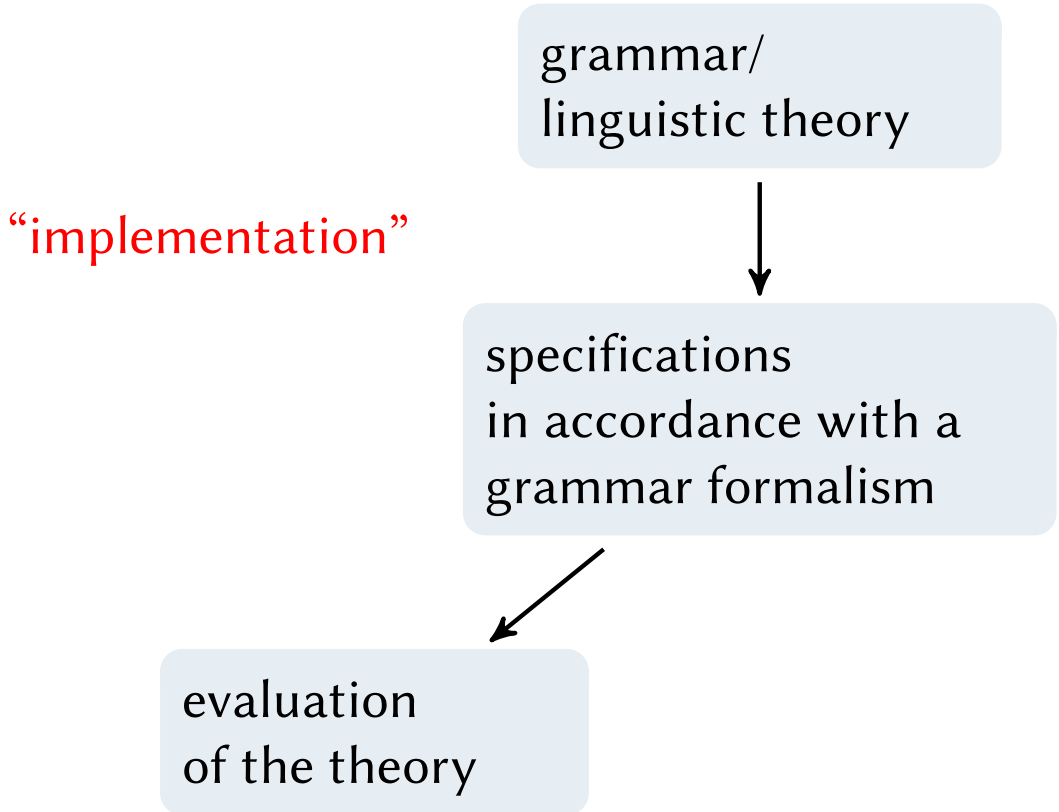
grammar sketches, example analyses
(incomplete)
↓
implemented grammars, digital resource
(complete)
↓
grammar in action, parsing
(i.a. usable in NLP)

Grammar engineering: the problem

- How to factorize the set of templates?
  - $\Rightarrow$ express lexical generalizations, e.g. active-passive diathesis
  - $\Rightarrow$ define tree families
- How to turn this into an electronic resource?
- How to plug it into a lexicon and use it?

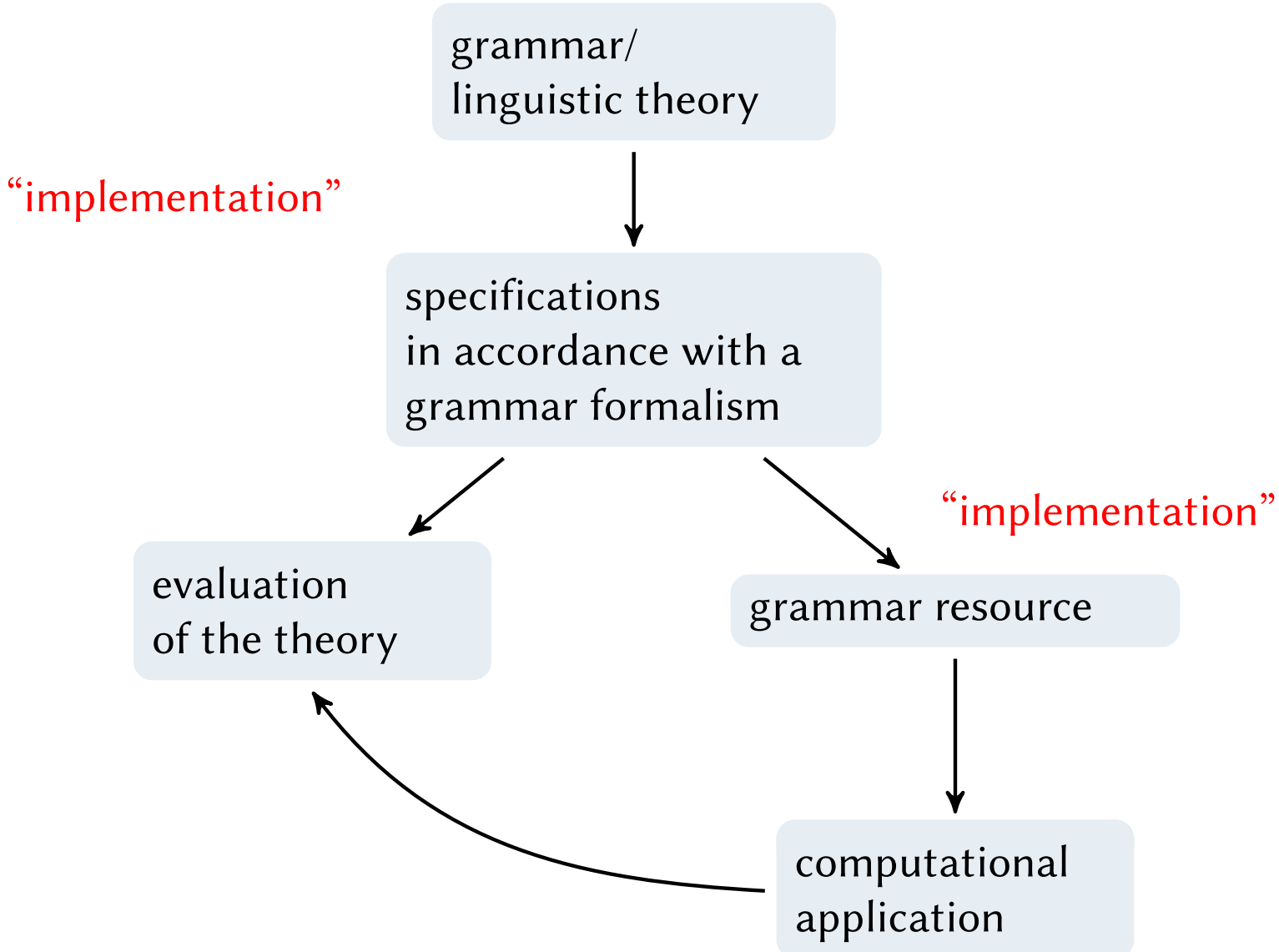
Outline

Two kinds of grammar implementation



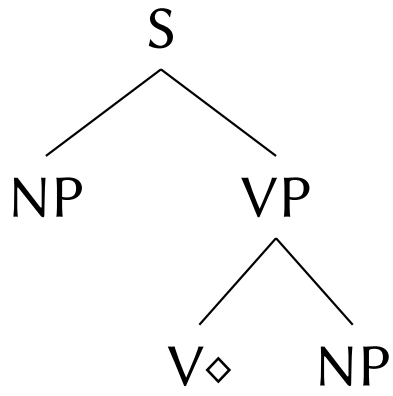
As is frequently pointed out but cannot be overemphasized, an important goal of formalization in linguistics is to enable subsequent researchers to see the defects of an analysis as clearly as its merits; only then can progress be made efficiently. (Dowty 1979: 322)

Two kinds of grammar implementation



What kind of grammar resource?

tree template



lexical insertion

anchor

repairs

The implementation task for LTAG

General task

Implement a large-coverage LTAG, i.e. based on the XTAG grammar!

Subtasks:

1. Generate unlexicalized trees (= tree templates)!
2. Generate a database of lexical anchors (= the lexicon)!
3. Connect the tree templates with the lexicon (= lexical insertion)!

Two ways of grammar implementation with TAG

Two existing toolkits:

XTAG tools[23]

1. implementation tools
   $\Rightarrow$ **metarule approach**
2. editor/viewer for MorphDB and SynDB
3. parser

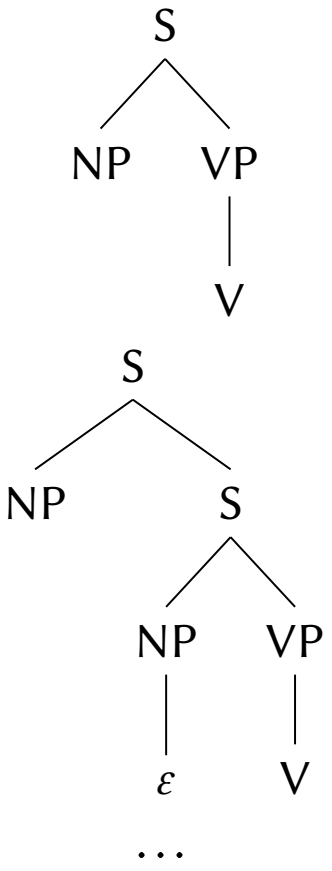
XMG + lexConverter + TuLiPA

1. XMG: eXtensible MetaGrammar[9]
   $\Rightarrow$ **metagrammar approach**
2. lexConverter (LEX2ALL)
3. TuLiPA: Tübingen Linguistic Parsing Architecture[16]

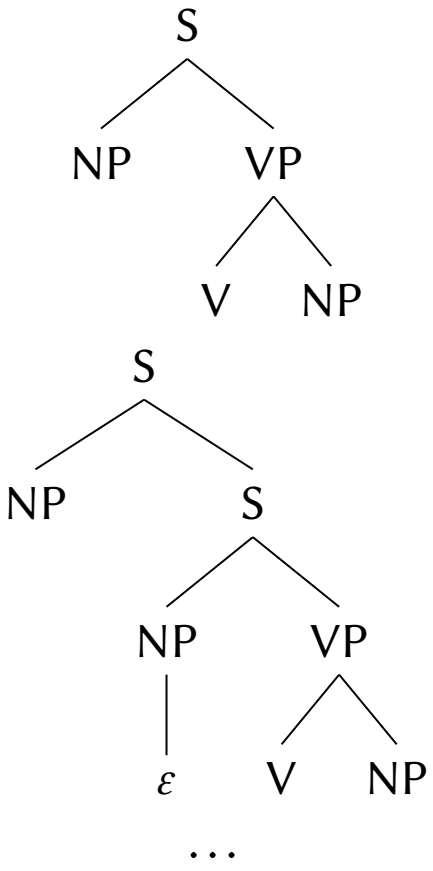
Outline

The situation

12 templates for intransitive verbs



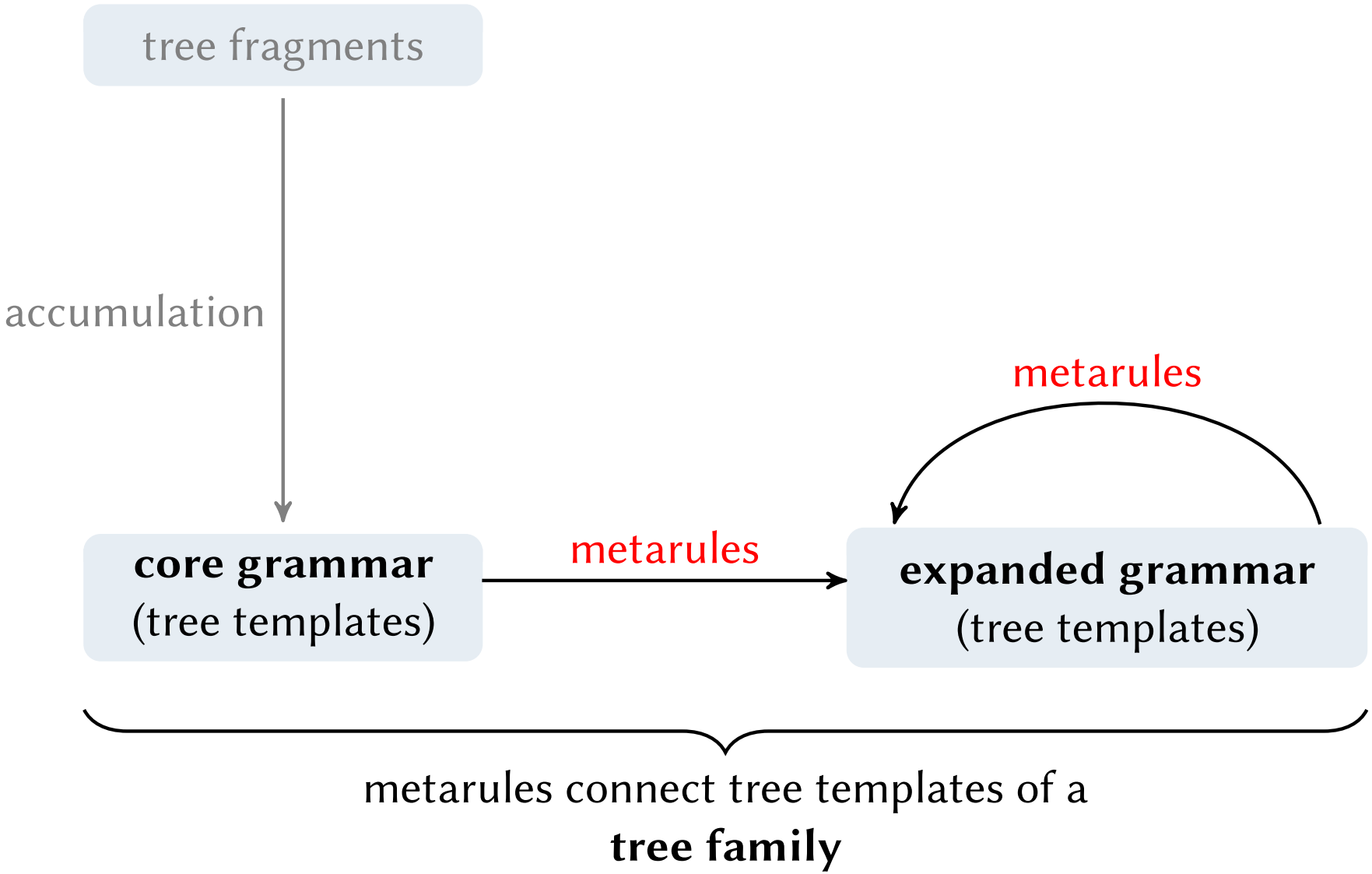
39 tree templates for transitive verbs



Basically, XTAG defines a set of 1008 unrelated tree templates.

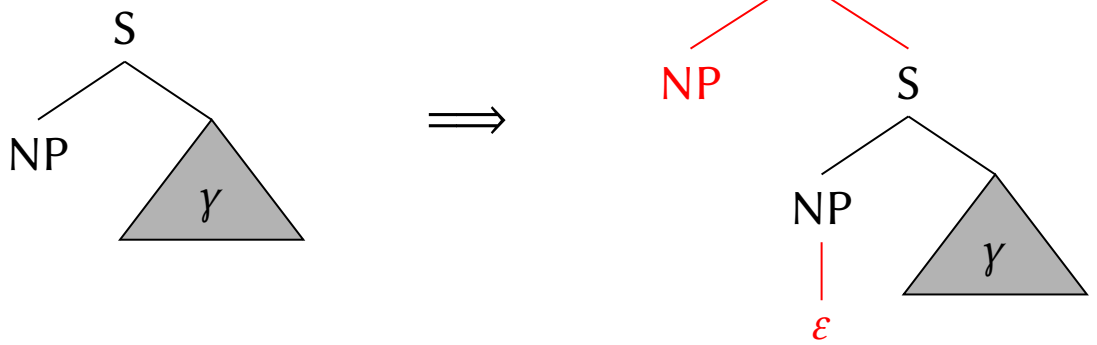
Metarules for LTAG

Idea from GPSG[12], later applied to XTAG[2,3,19]



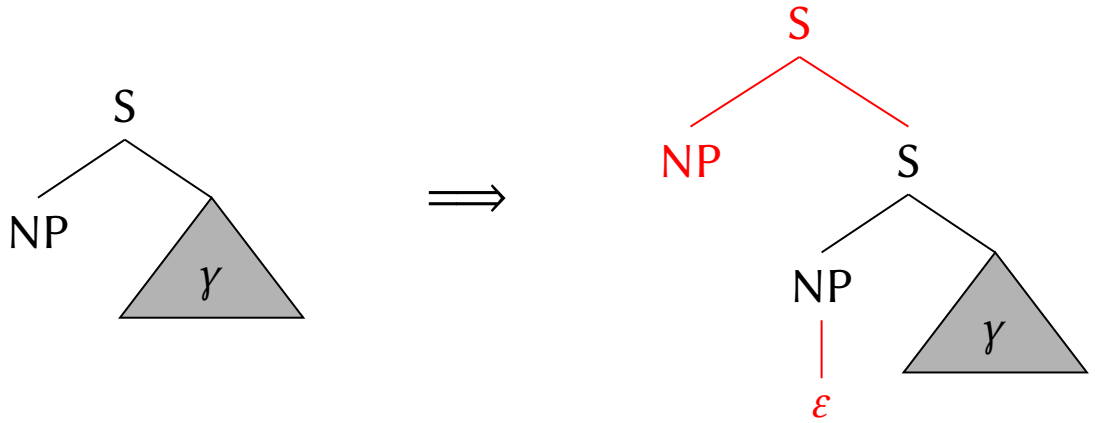
Metarules for LTAG: Example

extraction:

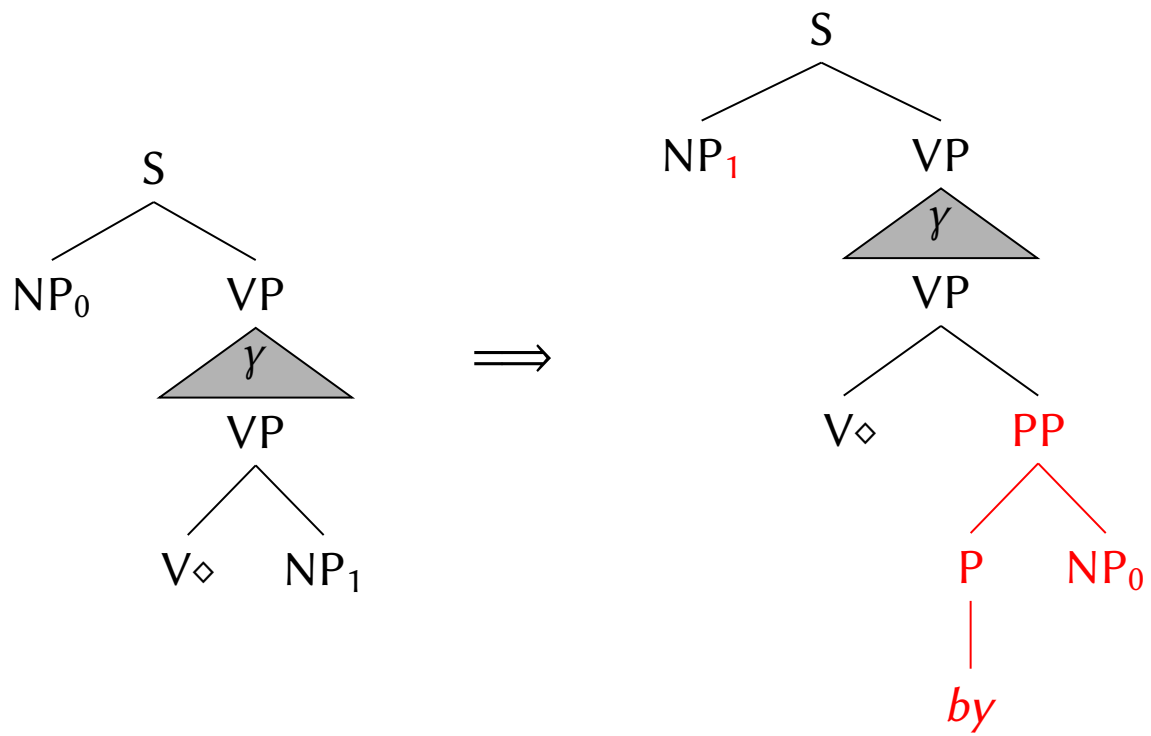


Metarules for LTAG: Example

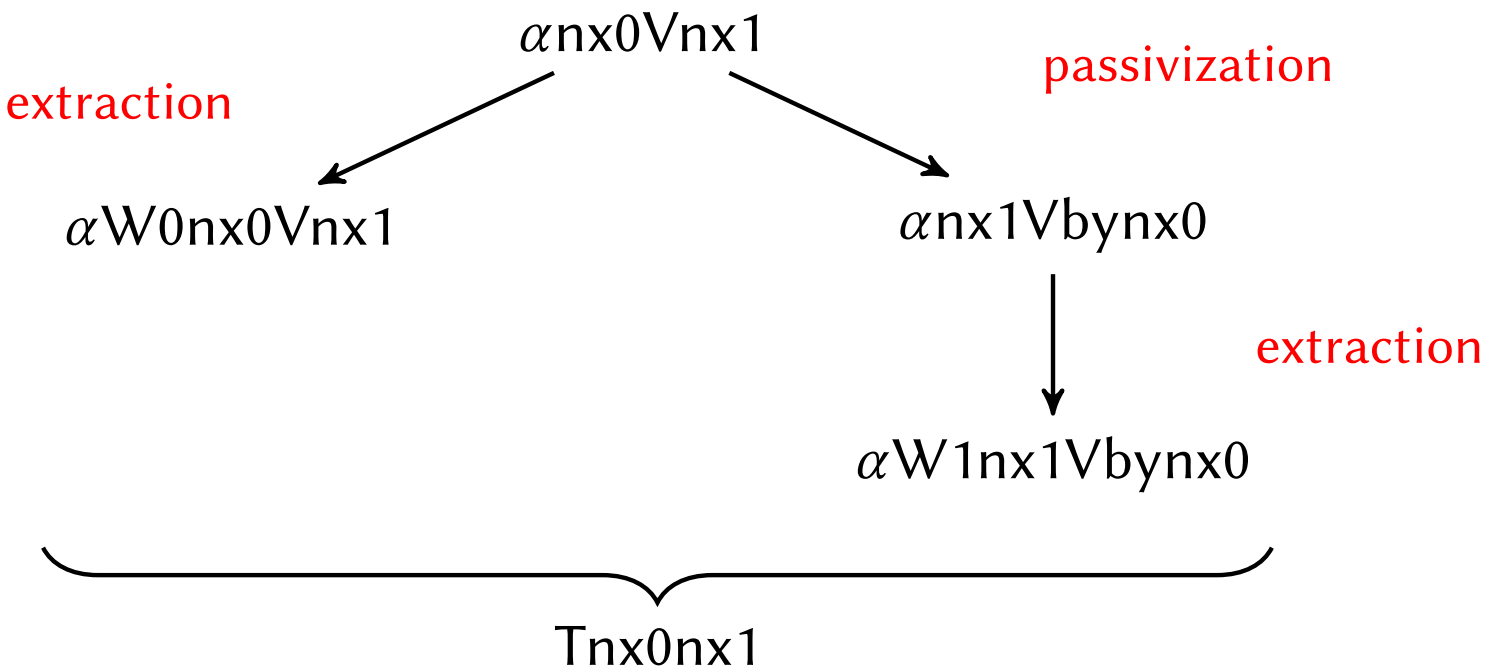
extraction:



passivization:



Metarules for LTAG: Example



Metarules for LTAG: Problems[2]

Metarules are very powerful:

- deletion, copying, recursive application, metavariables over trees
- order sensitive
- in the unrestricted case: undecidable[21]

Restrictions (GPSG):[20]

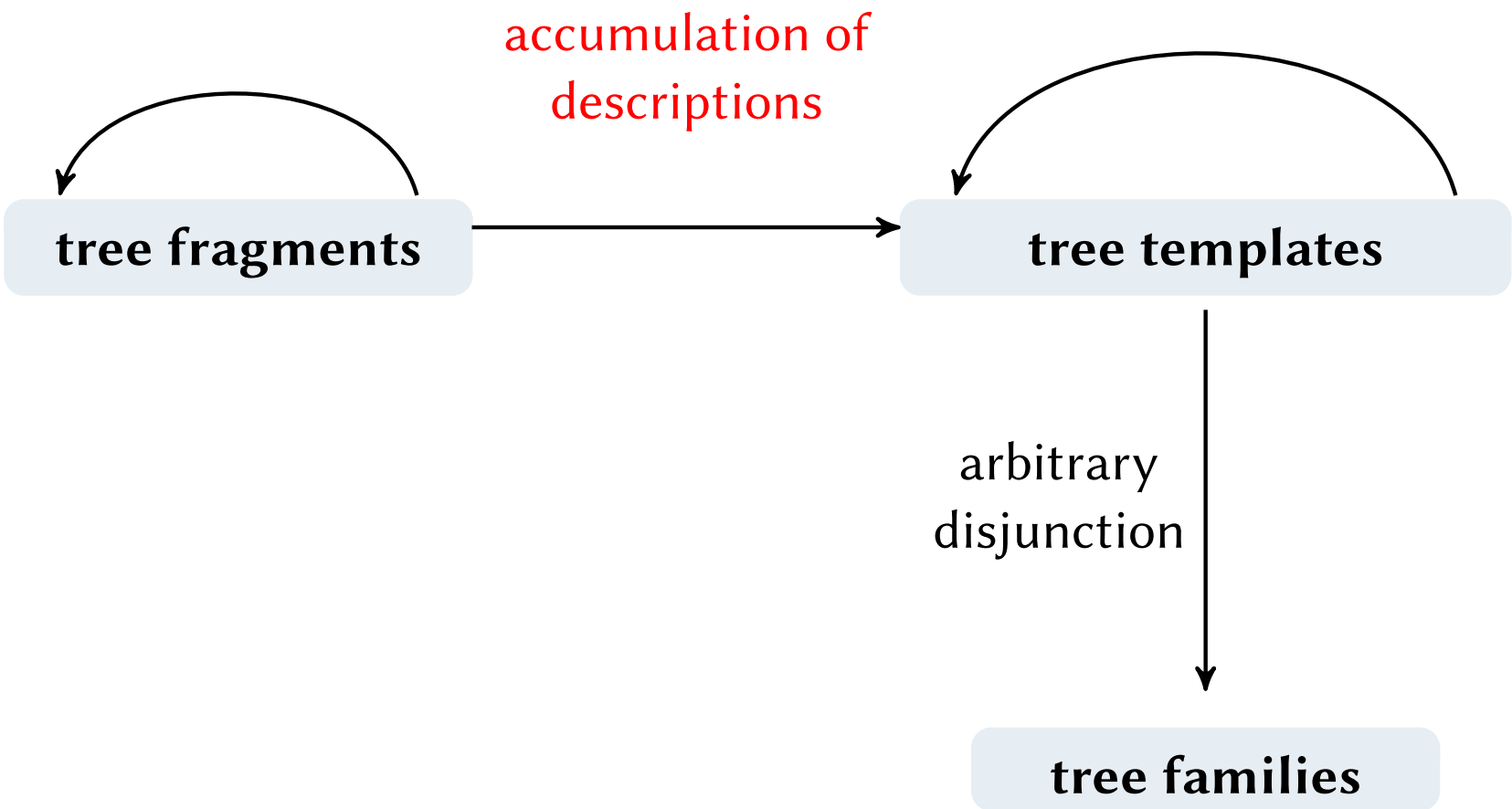
- finite closure: apply every metarule at most once!

  $\Rightarrow$ still NP-complete
- biclosure: apply at most two metarules in a row!

  $\Rightarrow$ insufficient for LTAG metarules[2]
- explicit rule ordering (by means of finite state automata)[19]

Metagrammars for LTAG

Candito (1996)[8,9,22]



Metagrammars for LTAG: Tree descriptions

$\mathcal{L}_D$: Description language for trees

Let n_1 and n_2 be node variables:

$$\mathit{Description} := \left(\begin{array}{lll} n_1 \rightarrow n_2 \mid & n_1 \rightarrow^+ n_2 \mid & n_1 \rightarrow^* n_2 \mid \\ n_1 \prec n_2 \mid & n_1 \prec^+ n_2 \mid & n_1 \prec^* n_2 \mid \\ n_1 = n_2 \mid & & \\ \mathit{Description} \wedge \mathit{Description} & & \end{array} \right)$$

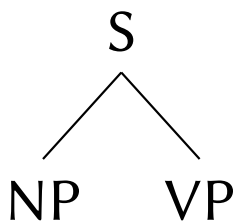
Metagrammars for LTAG: Tree descriptions

$\mathcal{L}_D$: Description language for trees

Let n_1 and n_2 be node variables:

$$\mathit{Description} := \left(\begin{array}{lll} n_1 \rightarrow n_2 \mid & n_1 \rightarrow^+ n_2 \mid & n_1 \rightarrow^* n_2 \mid \\ n_1 \prec n_2 \mid & n_1 \prec^+ n_2 \mid & n_1 \prec^* n_2 \mid \\ n_1 = n_2 \mid & & \\ \mathit{Description} \wedge \mathit{Description} & & \end{array} \right)$$

Example:



corresponds to

$$\begin{array}{ll} n_S \rightarrow n_{NP} & \wedge \\ n_S \rightarrow n_{VP} & \wedge \\ n_{NP} \prec n_{VP} & \end{array}$$

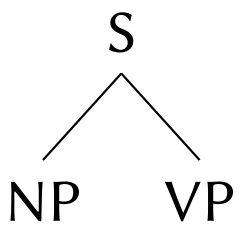
Metagrammars for LTAG: Tree descriptions

$\mathcal{L}_D$: Description language for trees

Let n_1 and n_2 be node variables:

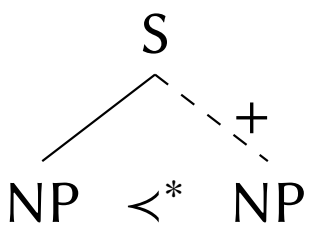
$$\mathit{Description} := \left(\begin{array}{lll} n_1 \rightarrow n_2 \mid & n_1 \rightarrow^+ n_2 \mid & n_1 \rightarrow^* n_2 \mid \\ n_1 \prec n_2 \mid & n_1 \prec^+ n_2 \mid & n_1 \prec^* n_2 \mid \\ n_1 = n_2 \mid & & \\ \mathit{Description} \wedge \mathit{Description} & & \end{array} \right)$$

Example:



corresponds to

$$\begin{array}{ll} n_S \rightarrow n_{NP} & \wedge \\ n_S \rightarrow n_{VP} & \wedge \\ n_{NP} \prec n_{VP} & \end{array}$$



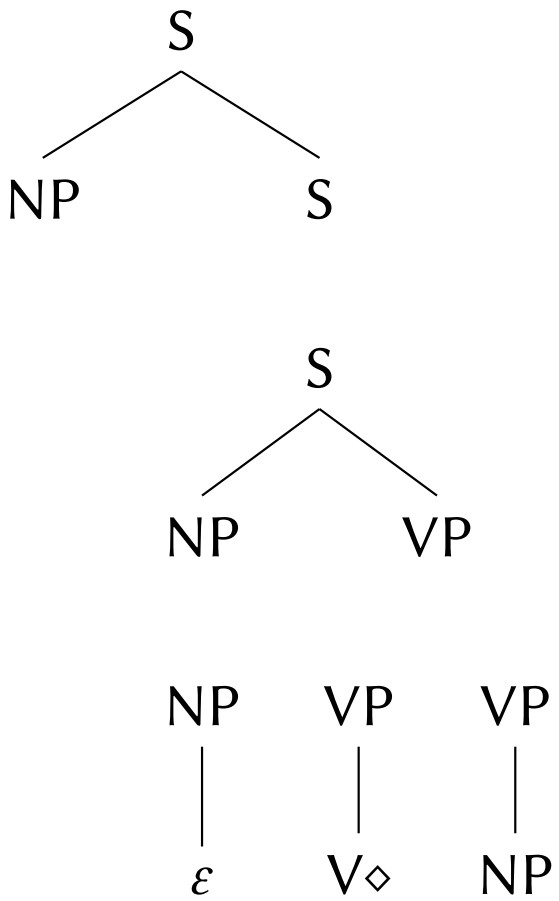
corresponds to

$$\begin{array}{ll} n_S \rightarrow n_{NP1} & \wedge \\ n_S \rightarrow^+ n_{NP2} & \wedge \\ n_{NP1} \prec^* n_{NP2} & \end{array}$$

Metagrammars for LTAG: Example

Minimal model of tree descriptions

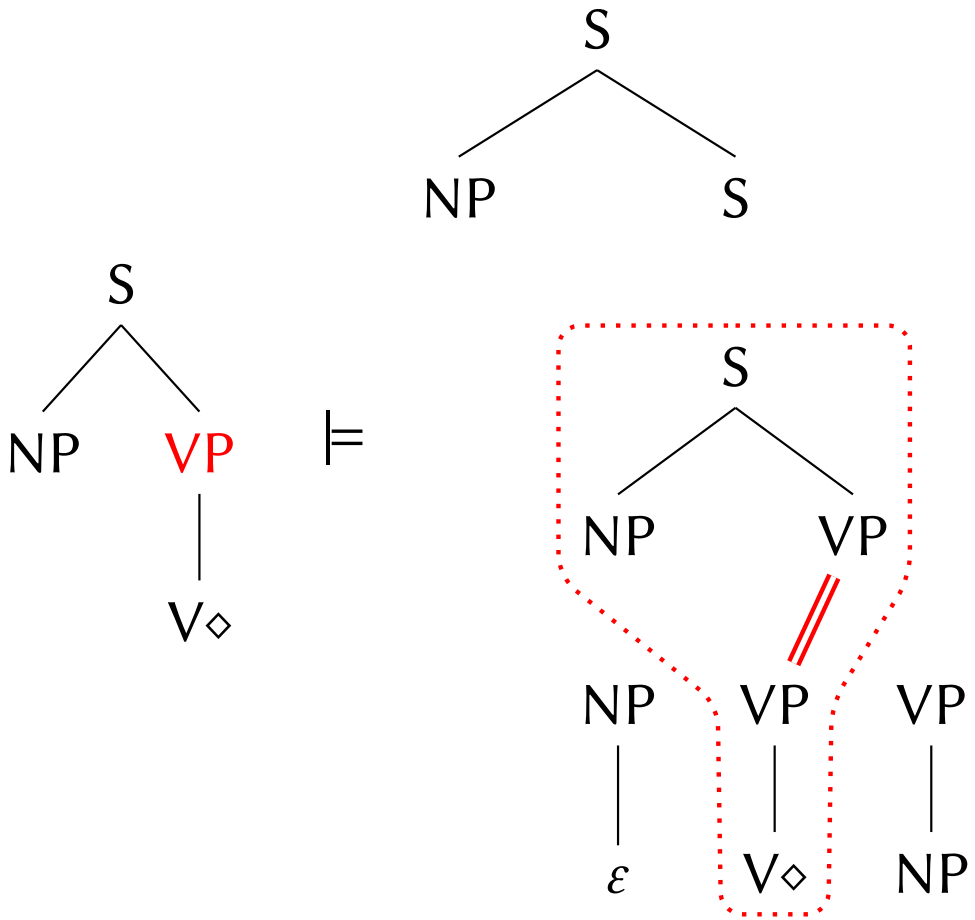
You may add edges but not nodes!



Metagrammars for LTAG: Example

Minimal model of tree descriptions

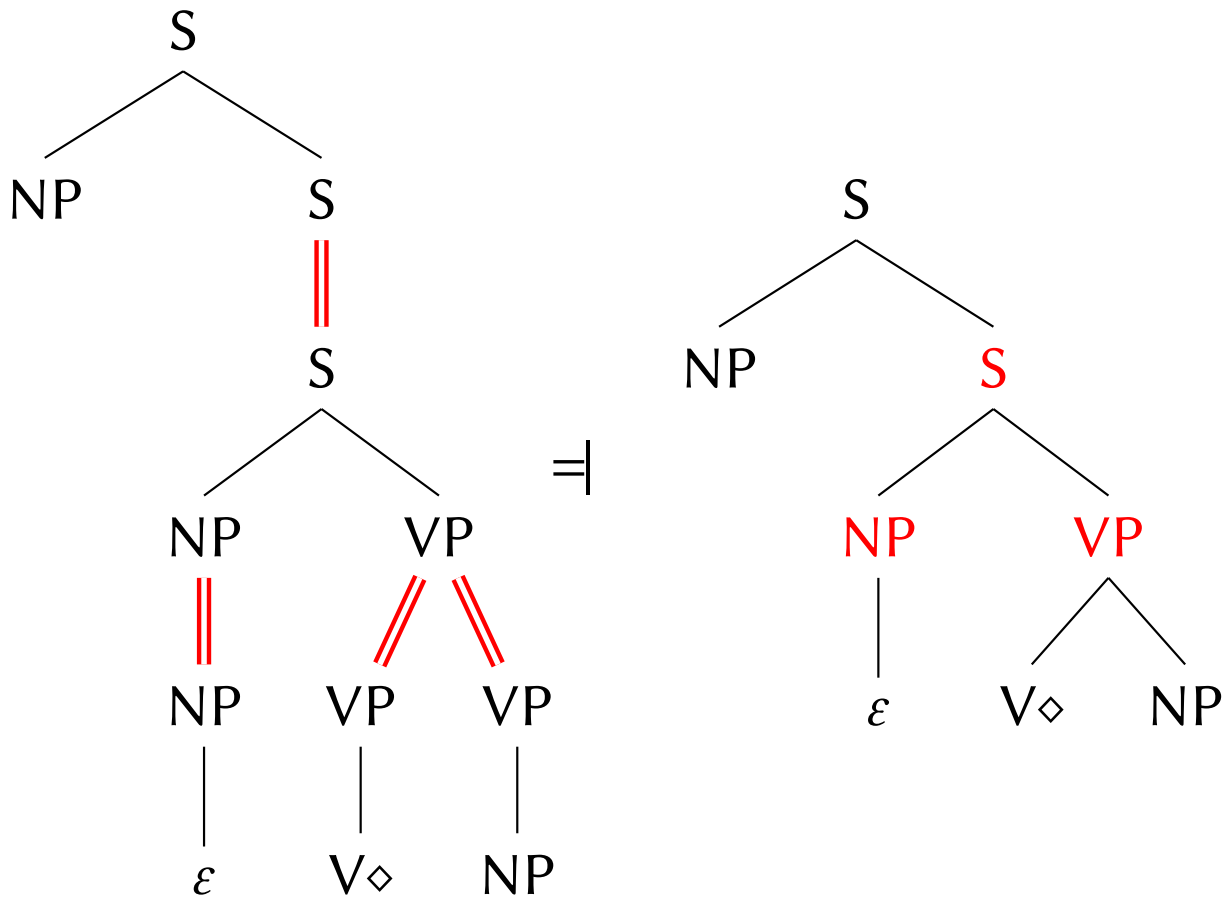
You may add edges but not nodes!



Metagrammars for LTAG: Example

Minimal model of tree descriptions

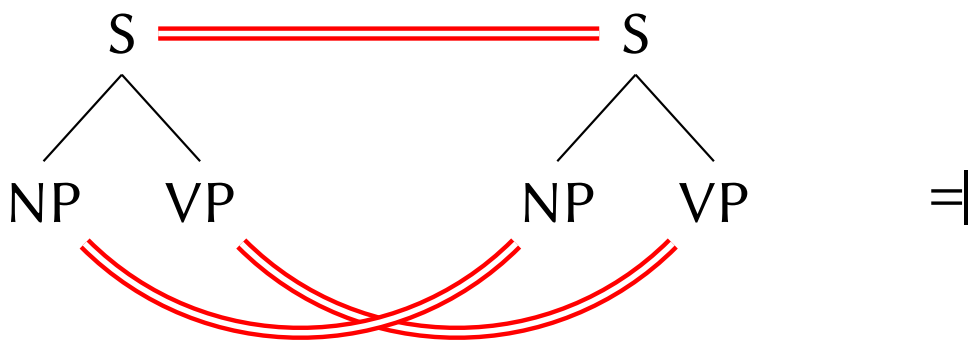
You may add edges but not nodes!



Metagrammars for LTAG: Example

Minimal model of tree descriptions

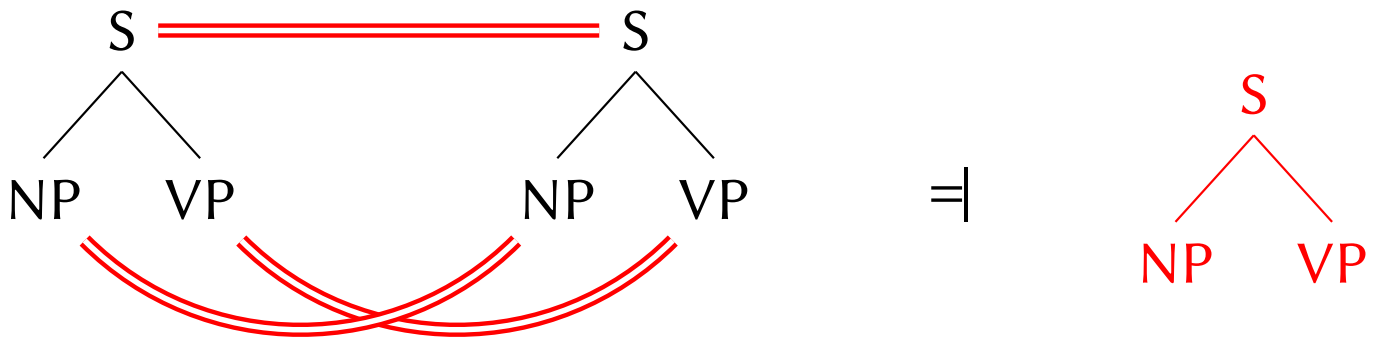
You may add edges but not nodes!



Metagrammars for LTAG: Example

Minimal model of tree descriptions

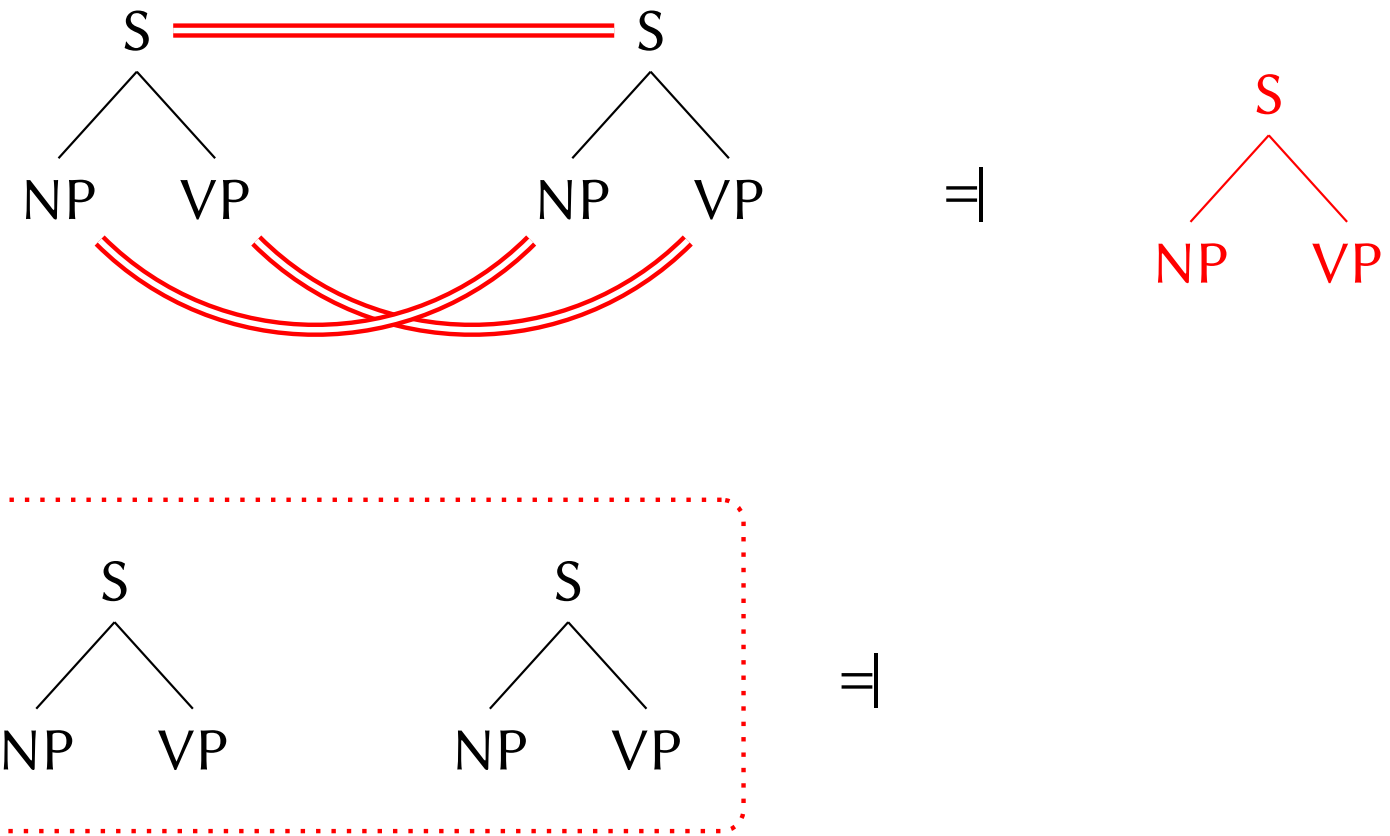
You may add edges but not nodes!



Metagrammars for LTAG: Example

Minimal model of tree descriptions

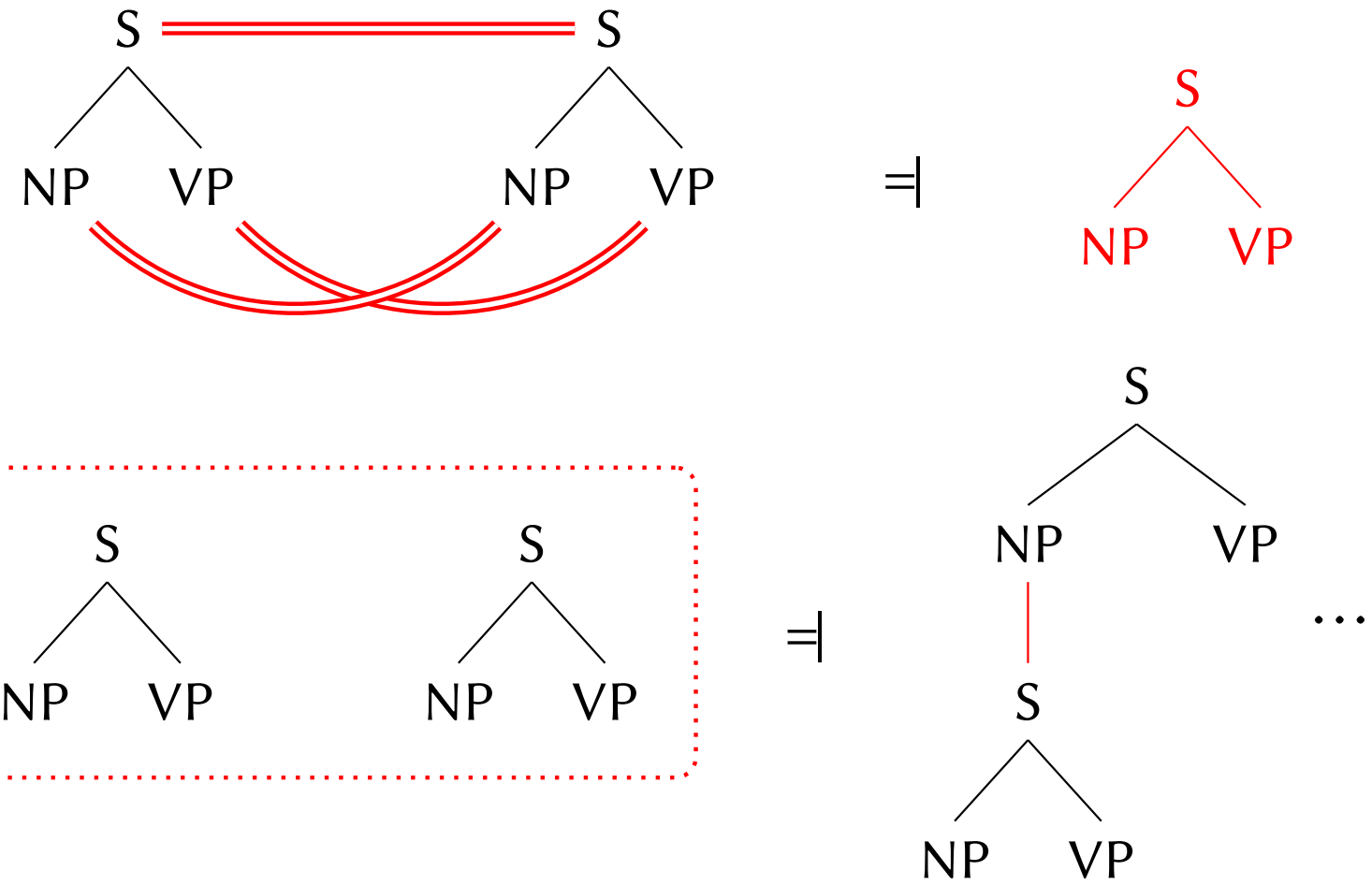
You may add edges but not nodes!



Metagrammars for LTAG: Example

Minimal model of tree descriptions

You may add edges but not nodes!

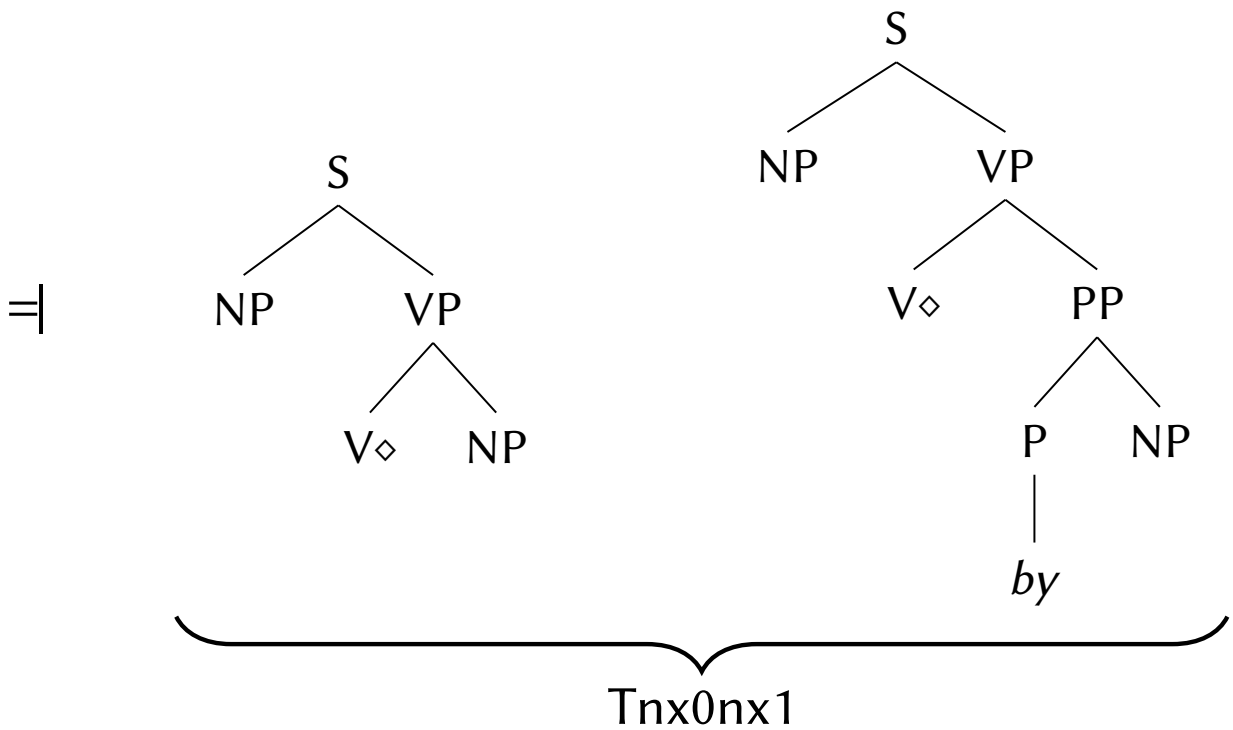


Metagrammars for LTAG: Properties

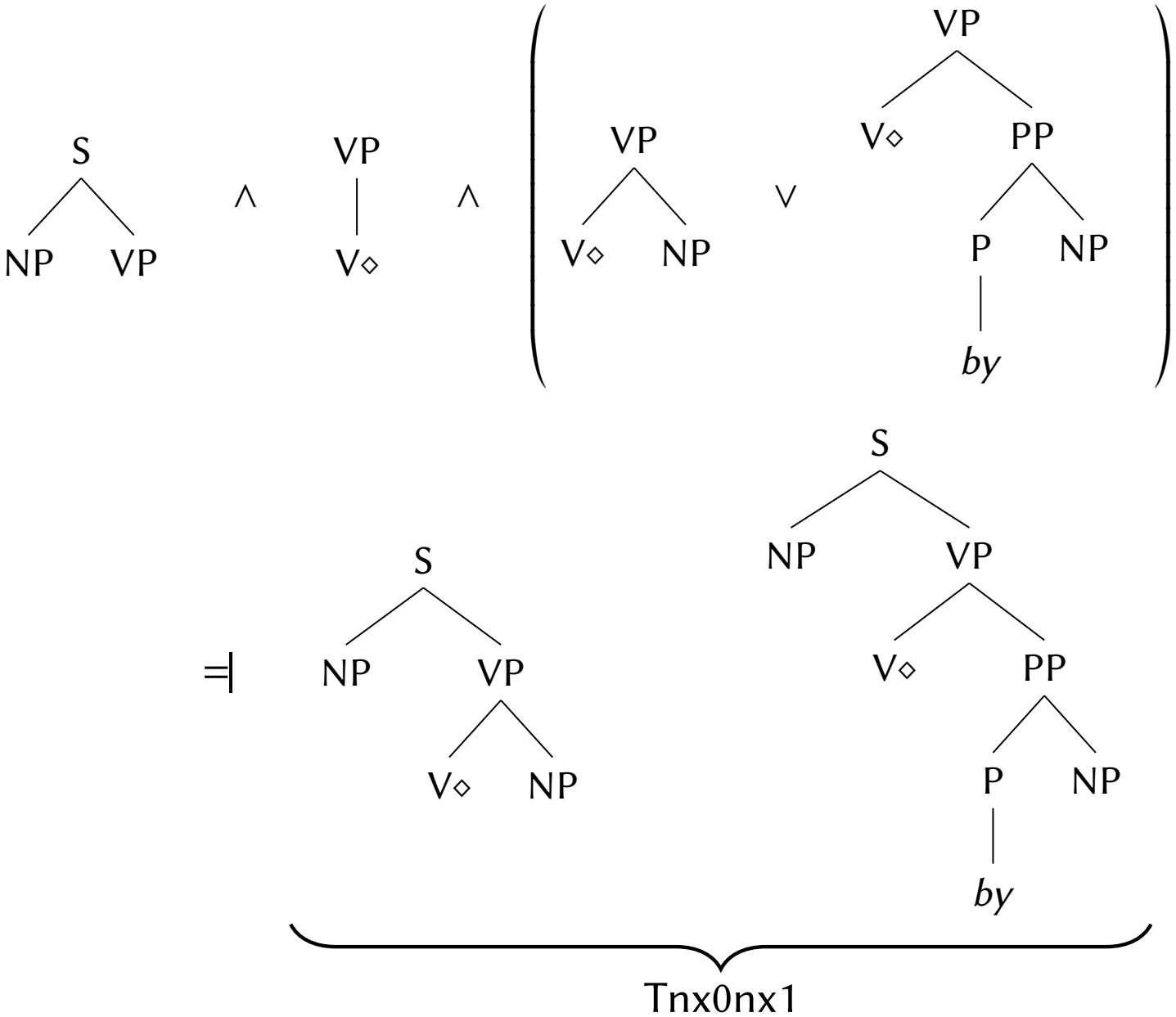
- no deletion, no copying, no recursion
- declarative, order insensitive
- The number of minimal models is finite.
- BUT: the number of minimal models can grow exponentially ($\mathcal{O}(n!)$) in terms of the number of described nodes.

Does it suffice? How to express passivization?

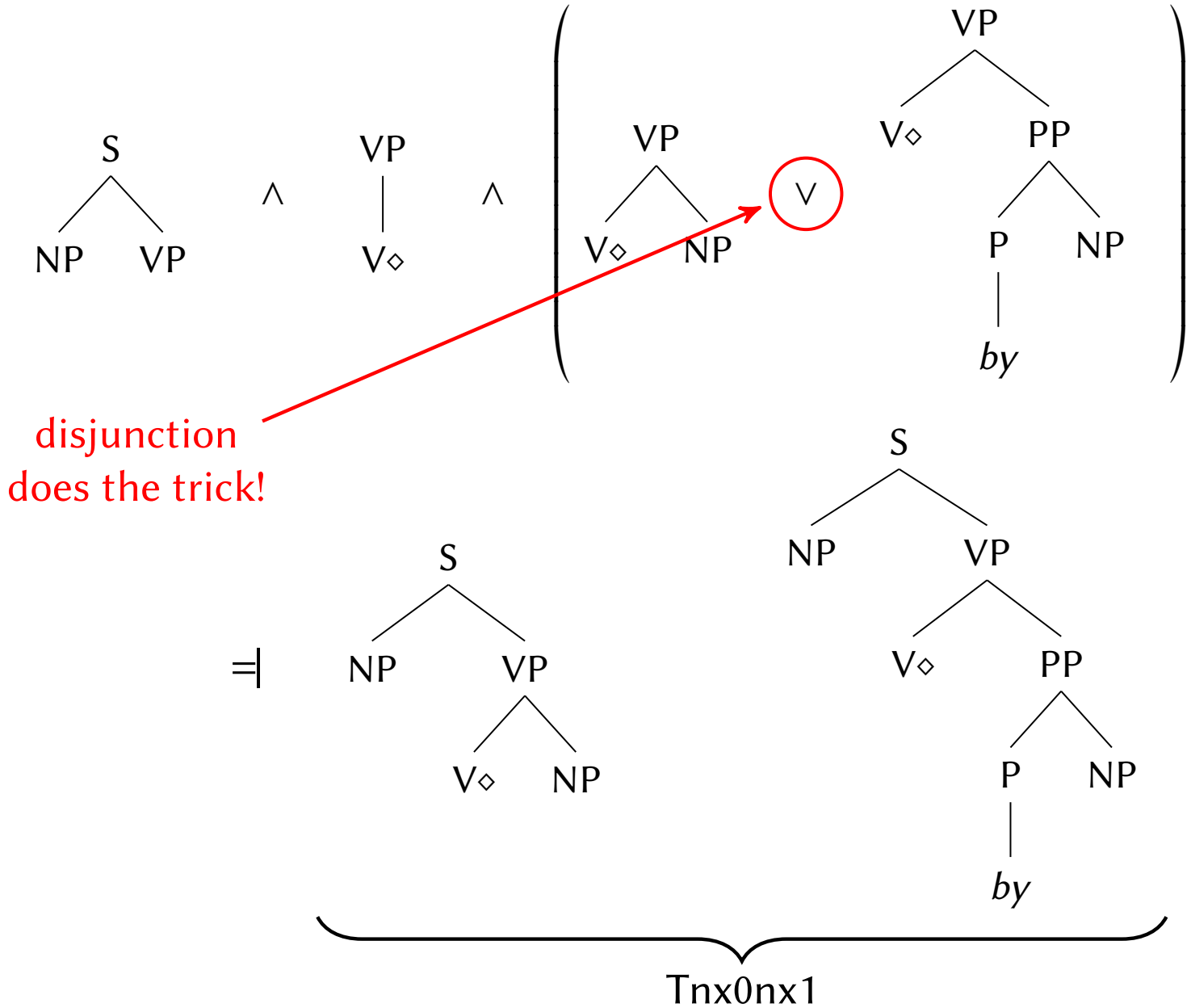
Metagrammars for LTAG: Passivization



Metagrammars for LTAG: Passivization



Metagrammars for LTAG: Passivization



Metagrammar for LTAG: Classes

Tree descriptions are bundled into so-called **classes**:

$\mathcal{L}_C$: Description language for the combination of tree descriptions

$$Class := Name : Content$$

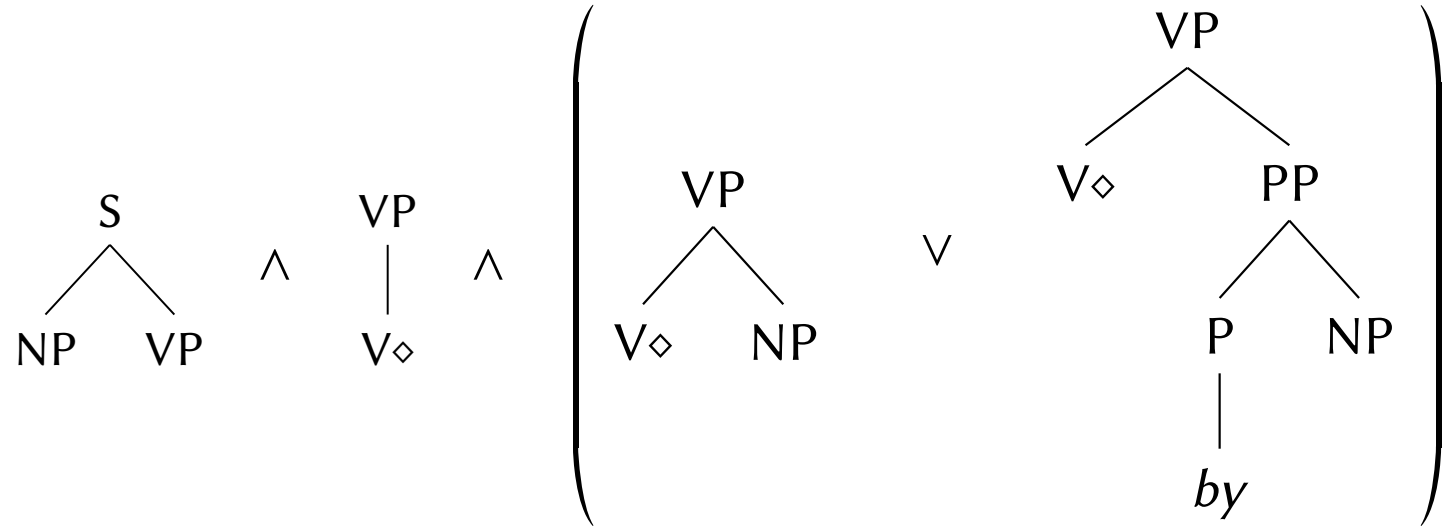
$$Content := \left(\begin{array}{l} Description \mid Name \mid \\ Content \vee Content \mid \\ Content \wedge Content \end{array} \right)$$

Upon instantiating/using a class:

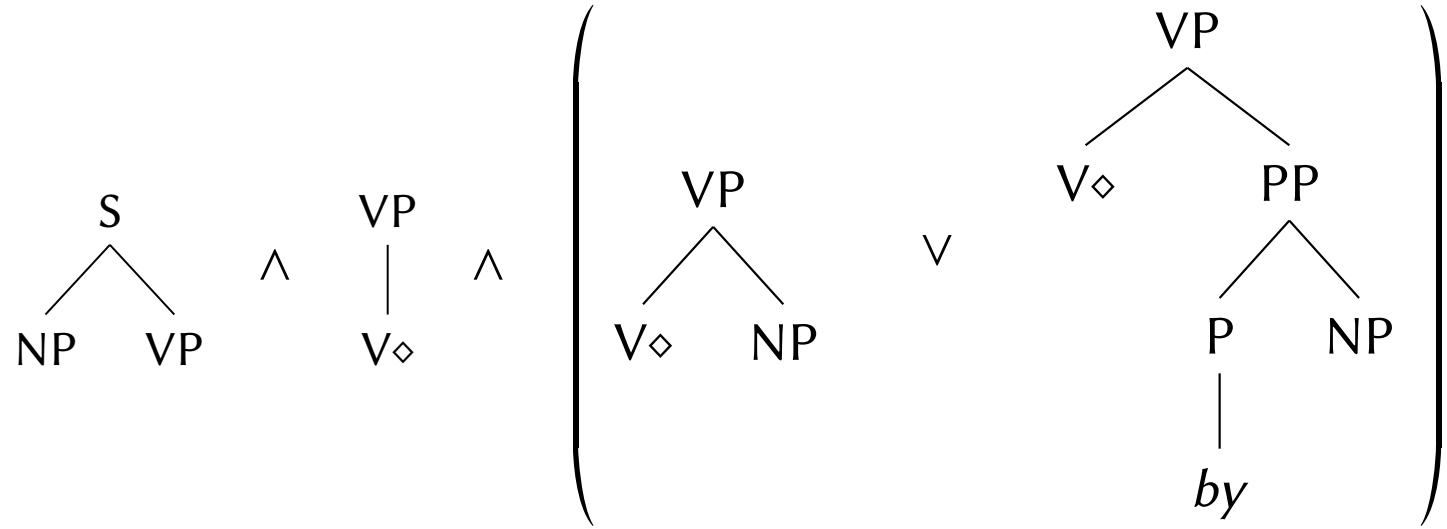
- Node variables are replaced by fresh ones.
- Node variables are known to the instantiating class.
- The class name is replaced by the content in the instantiating class.

$\Rightarrow$ Classes can be reused!

Metagrammar for LTAG: Classes

Tnx0Vnx1: 

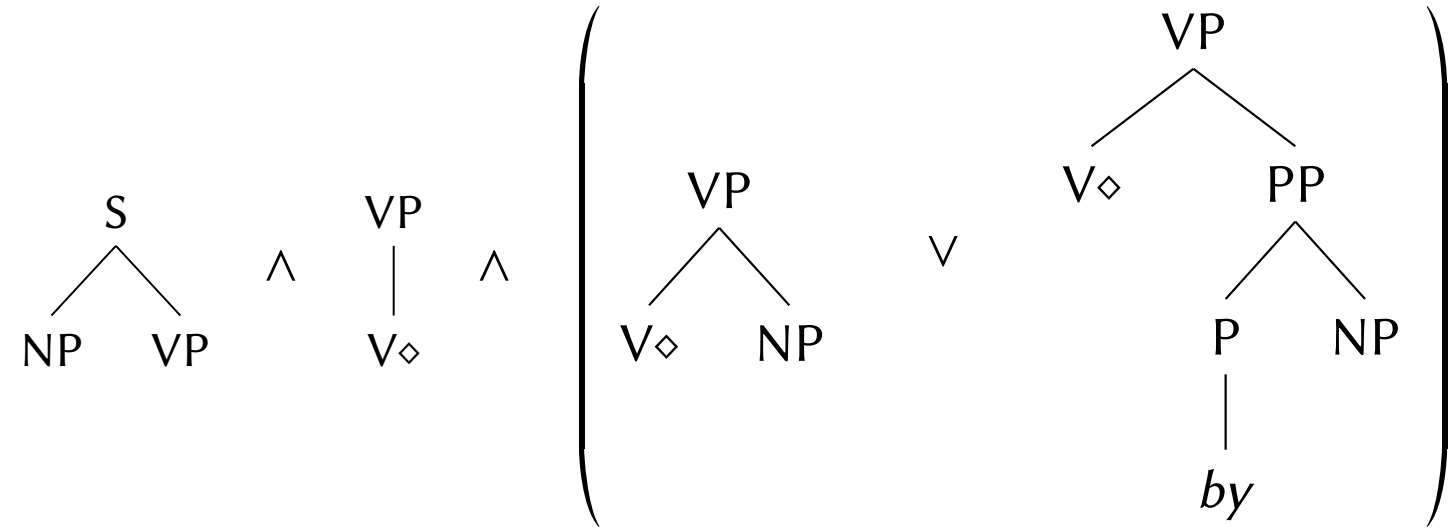
Metagrammar for LTAG: Classes

Tnx0Vnx1: 

Tnx0Vnx1: Subject ∧ VerbProjection ∧ (Object ∨ by-Phrase)

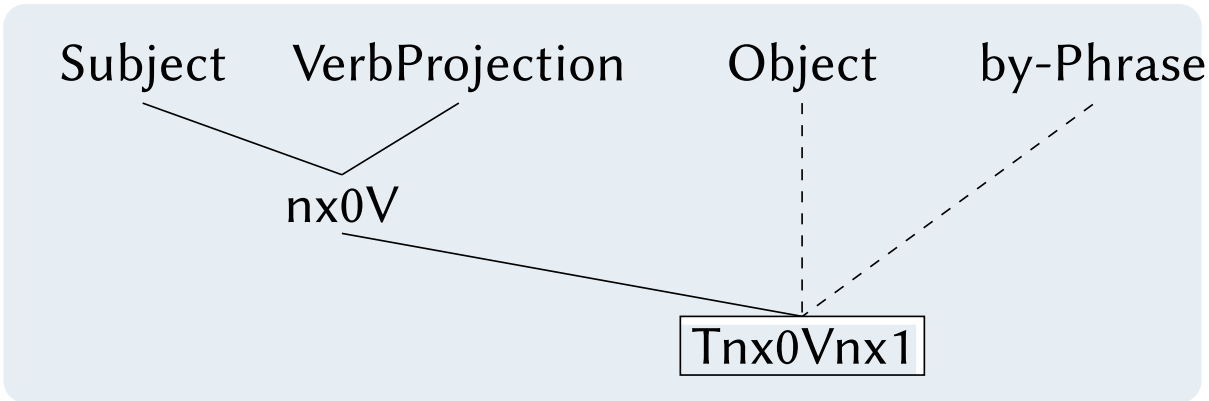
Tnx0Vnx1: nx0V ∧ (Object ∨ by-Phrase)

Metagrammar for LTAG: Classes

Tnx0Vnx1: 

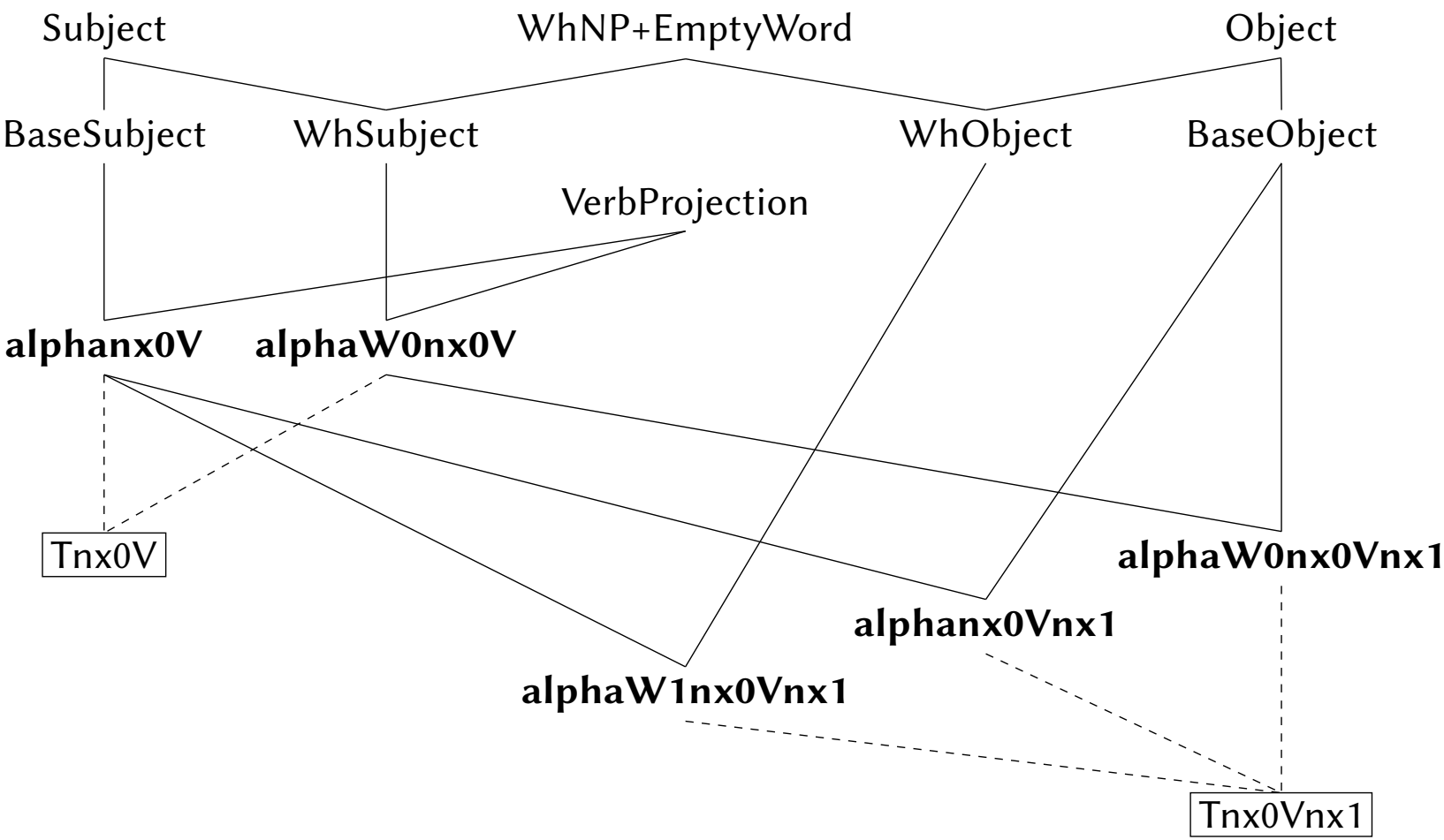
Tnx0Vnx1: Subject ∧ VerbProjection ∧ (Object ∨ by-Phrase)

Tnx0Vnx1: nx0V ∧ (Object ∨ by-Phrase)



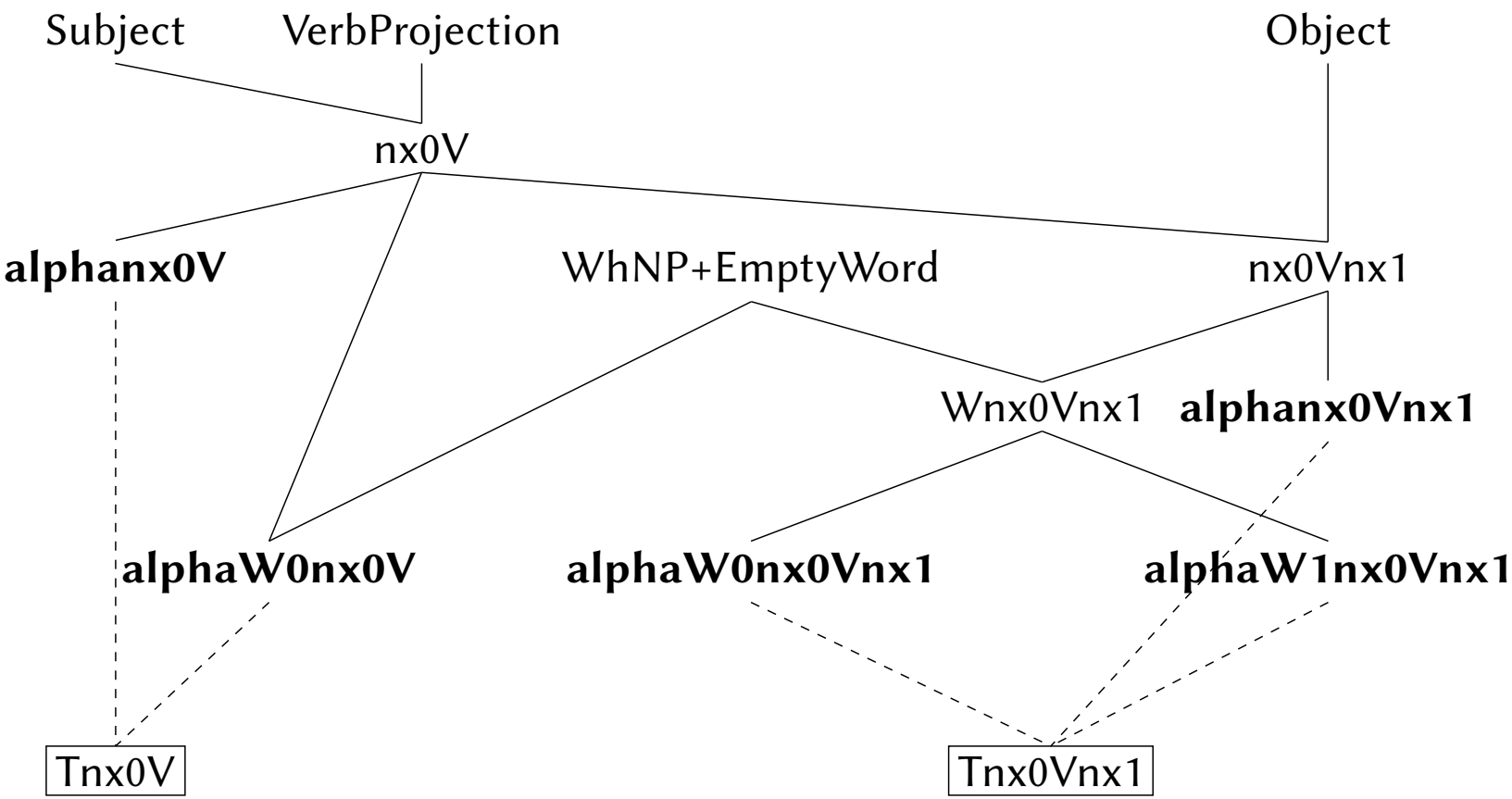
Metagrammar for LTAG: Class hierarchies

There are very many possible class hierarchies …



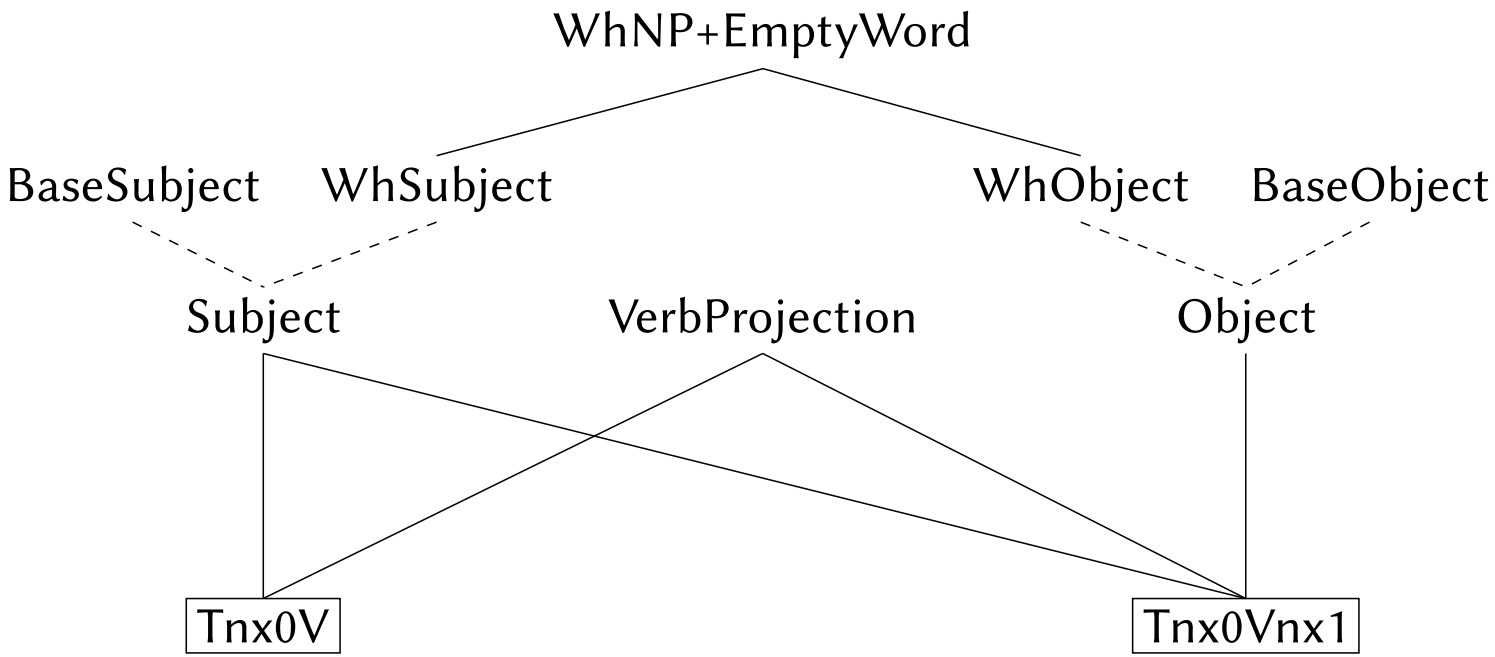
Metagrammar for LTAG: Class hierarchies

There are very many possible class hierarchies …



Metagrammar for LTAG: Class hierarchies

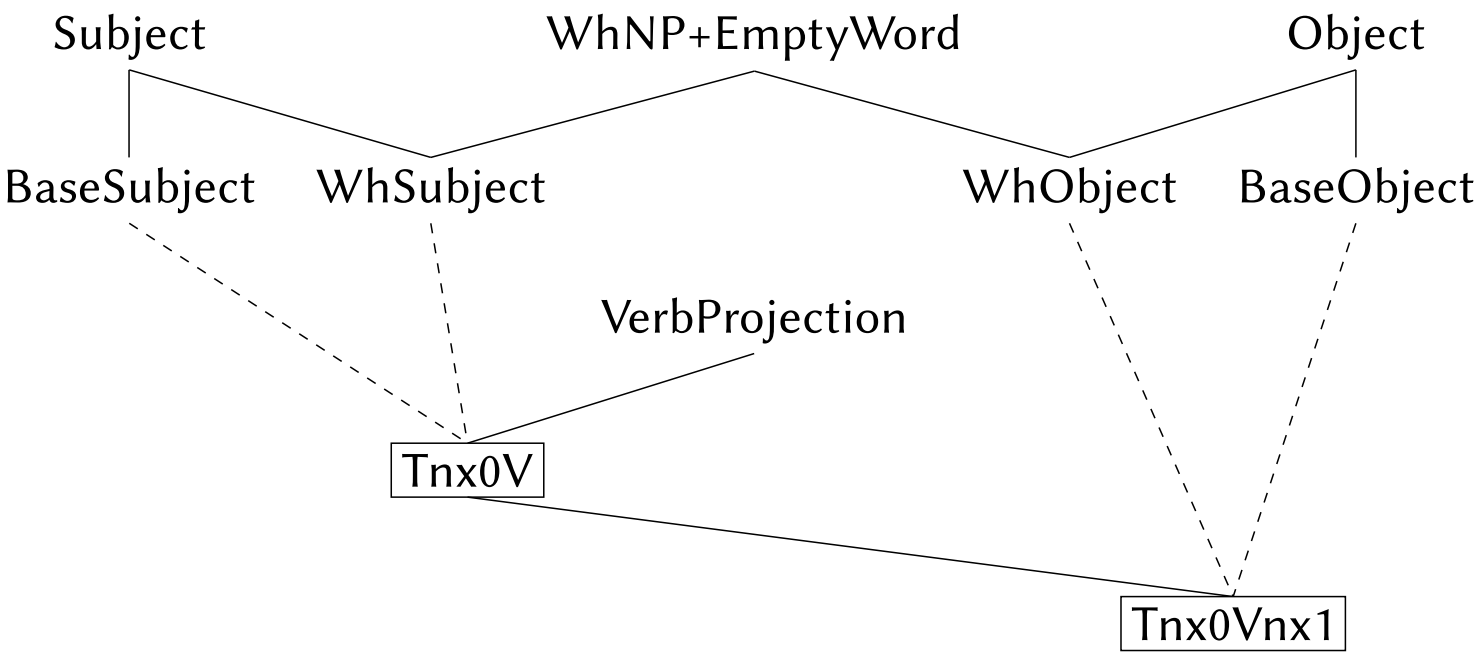
There are very many possible class hierarchies …



[1]

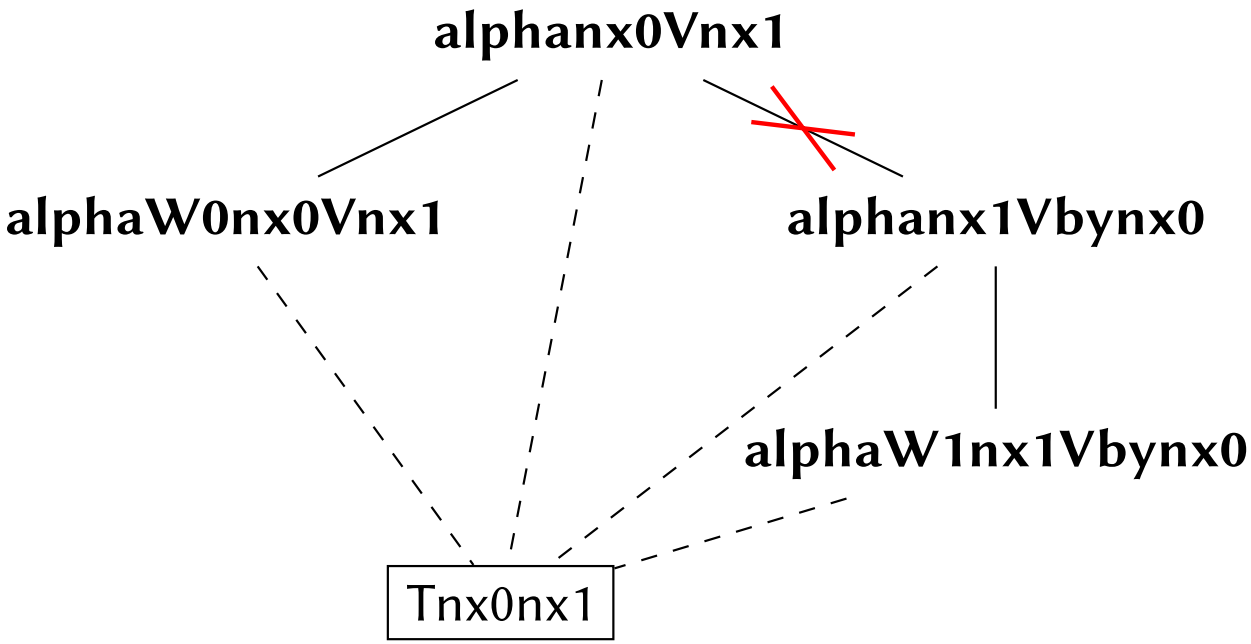
Metagrammar for LTAG: Class hierarchies

There are very many possible class hierarchies …



Metagrammar for LTAG: Class hierarchies

…but not everything is possible:



Outline

eXtensible Metagrammar (XMG): Background

- developed at LORIA, Nancy, LIFO, Orléans and HHU, Düsseldorf.[9]
- written in ~~Oz/Mozart~~ YAP and Python (as of XMG2)
- available at `dokufarm.phil.hhu.de/xmg`

eXtensible Metagrammar (XMG): Background

- developed at LORIA, Nancy, LIFO, Orléans and HHU, Düsseldorf.[9]
- written in ~~Oz/Mozart~~ YAP and Python (as of XMG2)
- available at `dokufarm.phil.hhu.de/xmg`

Why "eXtensible" ?

- highly modularized[17]
- dimensions with dedicated description languages and compilers (**<syn>**, **<sem>**, **<frame>**, **<morph>**, ...)
- interface using shared variables

eXtensible Metagrammar (XMG): Background

- developed at LORIA, Nancy, LIFO, Orléans and HHU, Düsseldorf.[9]
- written in ~~Oz/Mozart~~ YAP and Python (as of XMG2)
- available at `dokufarm.phil.hhu.de/xmg`

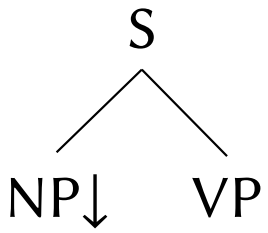
Why "eXtensible" ?

- highly modularized[17]
- dimensions with dedicated description languages and compilers (**<syn>**, **<sem>**, **<frame>**, **<morph>**, ...)
- interface using shared variables

Some existing implementations using XMG:

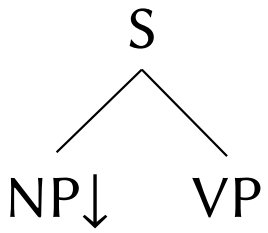
- French: FrenchTAG[8]
- English: XTAG with XMG[1]
- German: GerTT[14]

eXtensible Metagrammar (XMG): Example



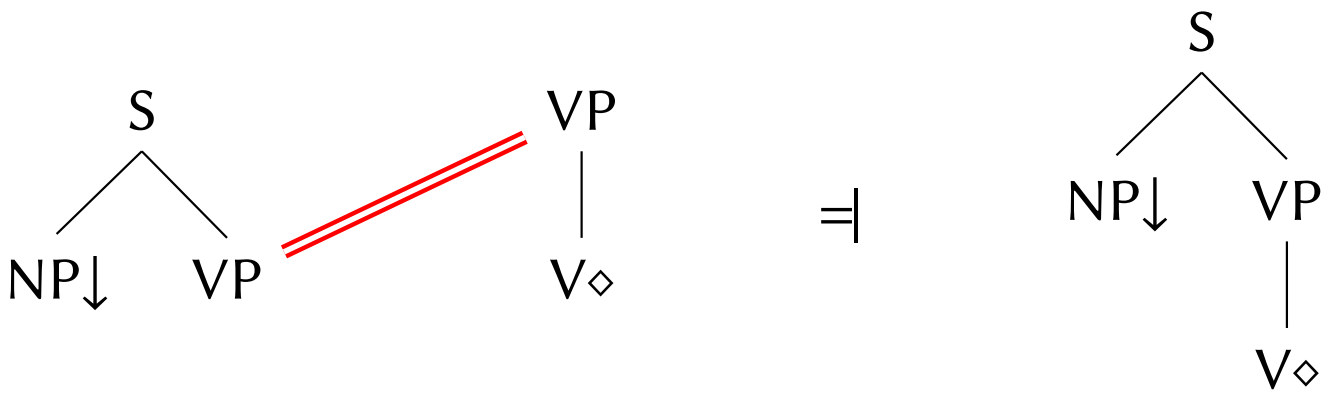

```
1  class Subject
2  export ?S
3  declare ?S ?NP ?VP
4  { <syn>{
5      node ?S [cat=s]{
6          node ?NP (mark=subst) [cat=np]
7          node ?VP [cat=vp]
8          }
9      }
10 }
```

eXtensible Metagrammar (XMG): Example




```
1  class Subject
2  export ?S
3  declare ?S ?NP ?VP
4  { <syn>{
5      node ?S [cat=s];
6      node ?NP (mark=subst) [cat=np];
7      node ?VP [cat=vp];
8      ?S -> ?NP;
9      ?S -> ?VP;
10     ?NP >> ?VP
11     }
12 }
```

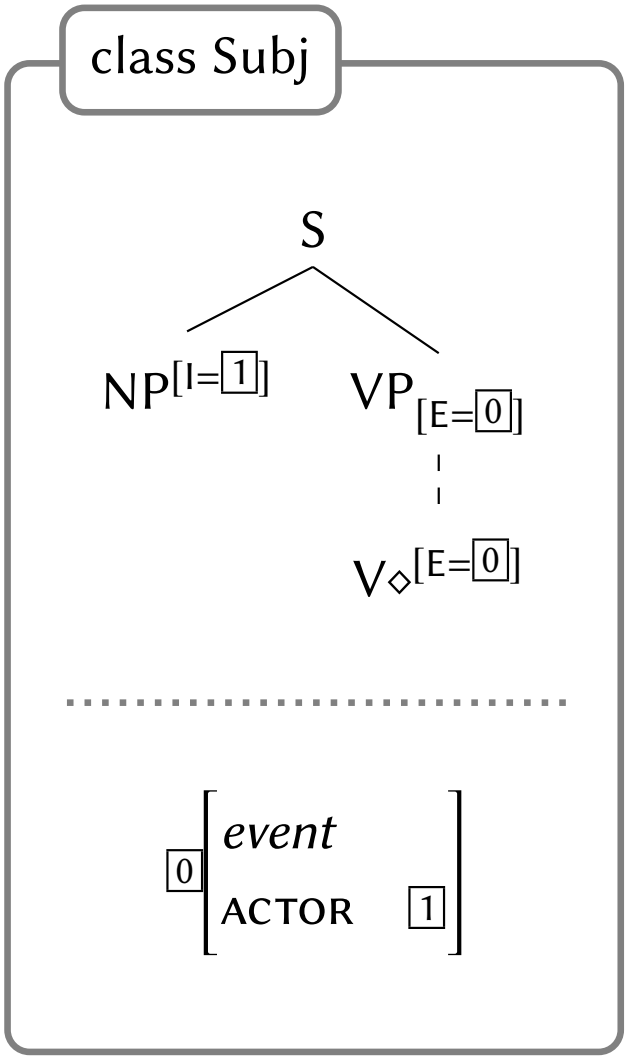
eXtensible Metagrammar (XMG): Example




```
1  class alphanx0v
2  import VerbProjection[]
3  declare ?Subj
4  {
5    ?Subj = Subject[];
6    ?Subj.?VP = ?VP
7  }
```

eXtensible Metagrammar (XMG): Example with <frame>

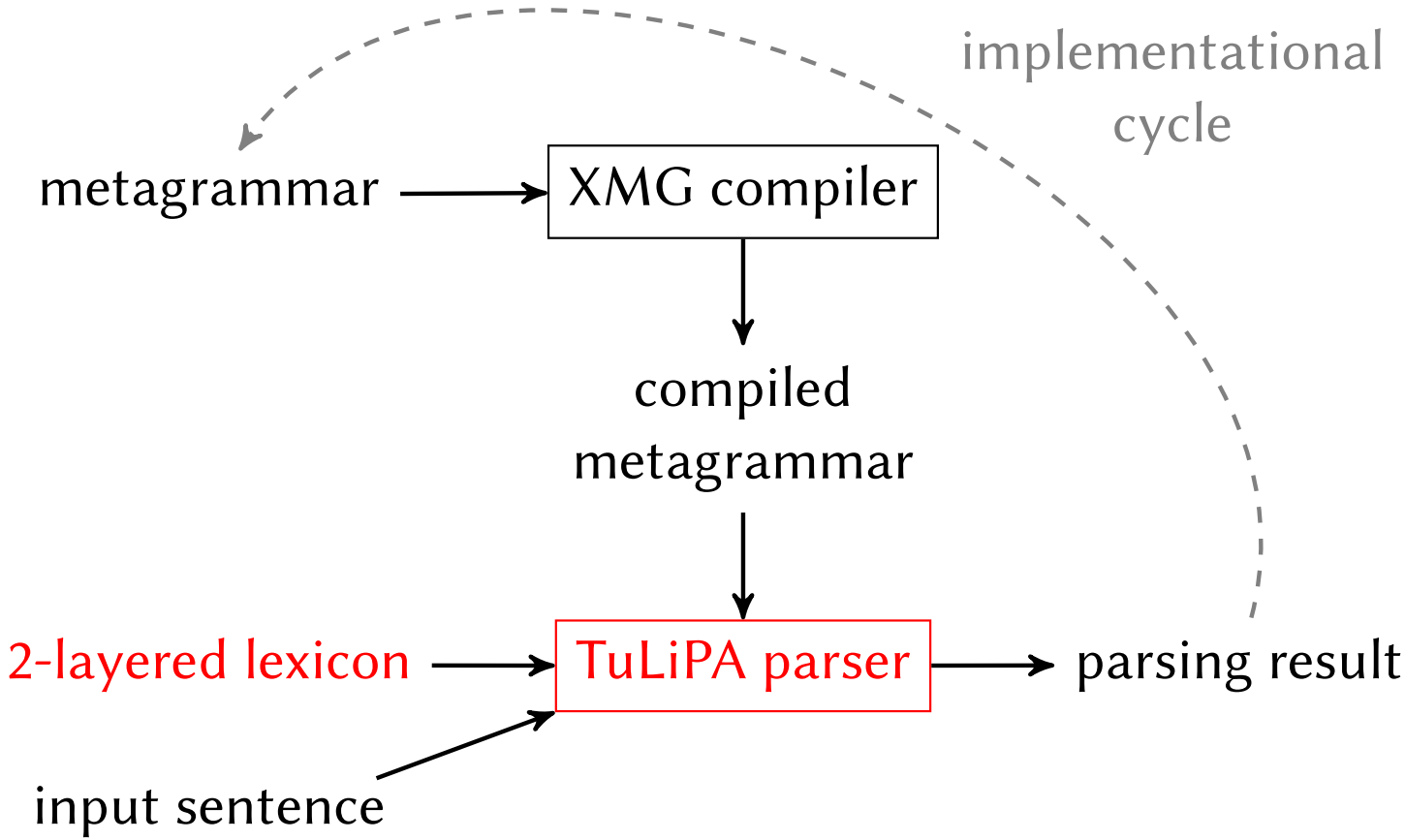
(Lichte & Petitjean)[15]



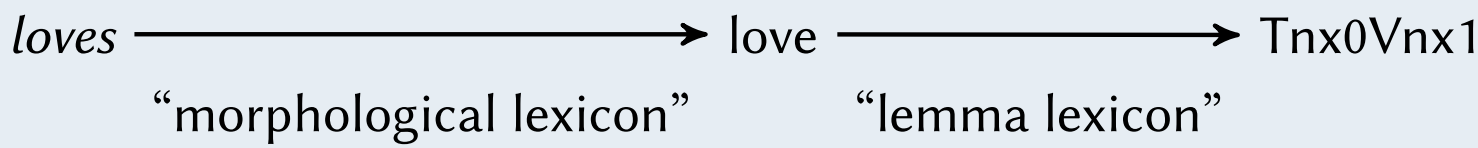

```
class Subj
...
<syn>{
  node ?S [cat=s];
  node ?SUBJ [cat=np,
              top=[i=?1]];
  node ?VP [cat=vp,bot=[e=?0]];
  node ?V (mark=anchor)
          [cat=v,top=[e=?0]];
  ?S -> ?SUBJ; ?S -> ?VP; ?VP -> * ?V;
  ?SUBJ >> ?VP
};
<frame>{
?0[event,
    actor:?1]
}
...
```

Outline

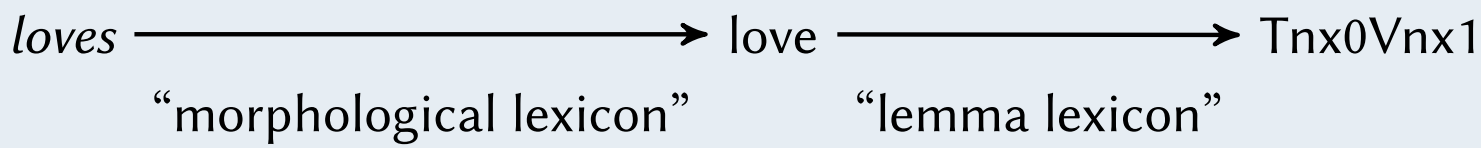
Lexicon and parser



Lexicon and parser: A 2-layered lexicon



Lexicon and parser: A 2-layered lexicon



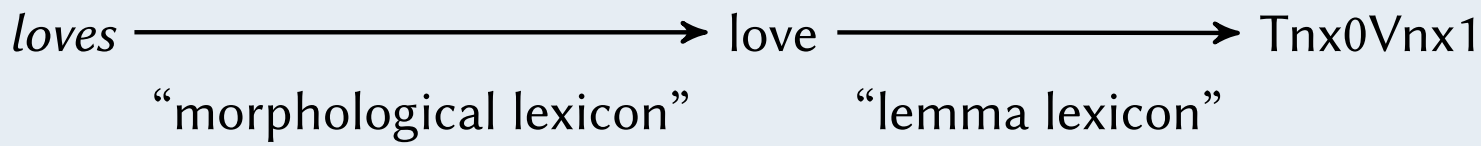
Morphological lexicon

maps an (inflected) token to some base form (= lemma), while preserving morphological information in a feature structure.

loves	love	[pos=v; num=sing; pers=3;]
Peter	Peter	[pos=n; num=sing; pers=3; case=nom|acc;]

Interface with tree templates: Feature unification during lexical insertion

Lexicon and parser: A 2-layered lexicon



Lemma lexicon

maps a lemma onto tree tuple families, while also containing selectional restrictions (e.g., case assignment).

```
*ENTRY: love
*CAT: v
*SEM:
*ACC: 1
*FAM: Tnx0Vnx1
*FILTERS: []
*EX:
*EQUATIONS:
NParg1 -> case = nom
NParg2 -> case = acc
*COANCHORS:
```

Interface with tree templates:
EQUATIONS → nodes of tree templates
FILTERS → selection of tree templates

Lexicon and parser: The TuLiPA parser

(Parmentier et al.)[16]

TuLiPA

- Tübingen Linguistic Parsing Architecture (TuLiPA)
- uses Range Concatenation Grammar (RCG) as a pivot formalism.

Components:

1. TAG-to-RCG converter (on-line)
2. RCG parser → RCG derivation forest → TAG derivation forest
3. Parse viewer (derived tree, derivation tree, dependency view, semantic representation)

Availability of TuLiPA:
written in Java and released under the GNU GPL
(`http://sourcesup.cru.fr/tulipa/`)

Outline

How does it work?

XMG processing steps are as follow:

- The metagrammar is compiled: metagrammatical language is translated into executable code
- The generated code is executed: accumulation of descriptions into the dimensions
- Descriptions are solved: every dimension comes with a dedicated solver
- Models are converted into the output language (XML)

Tools

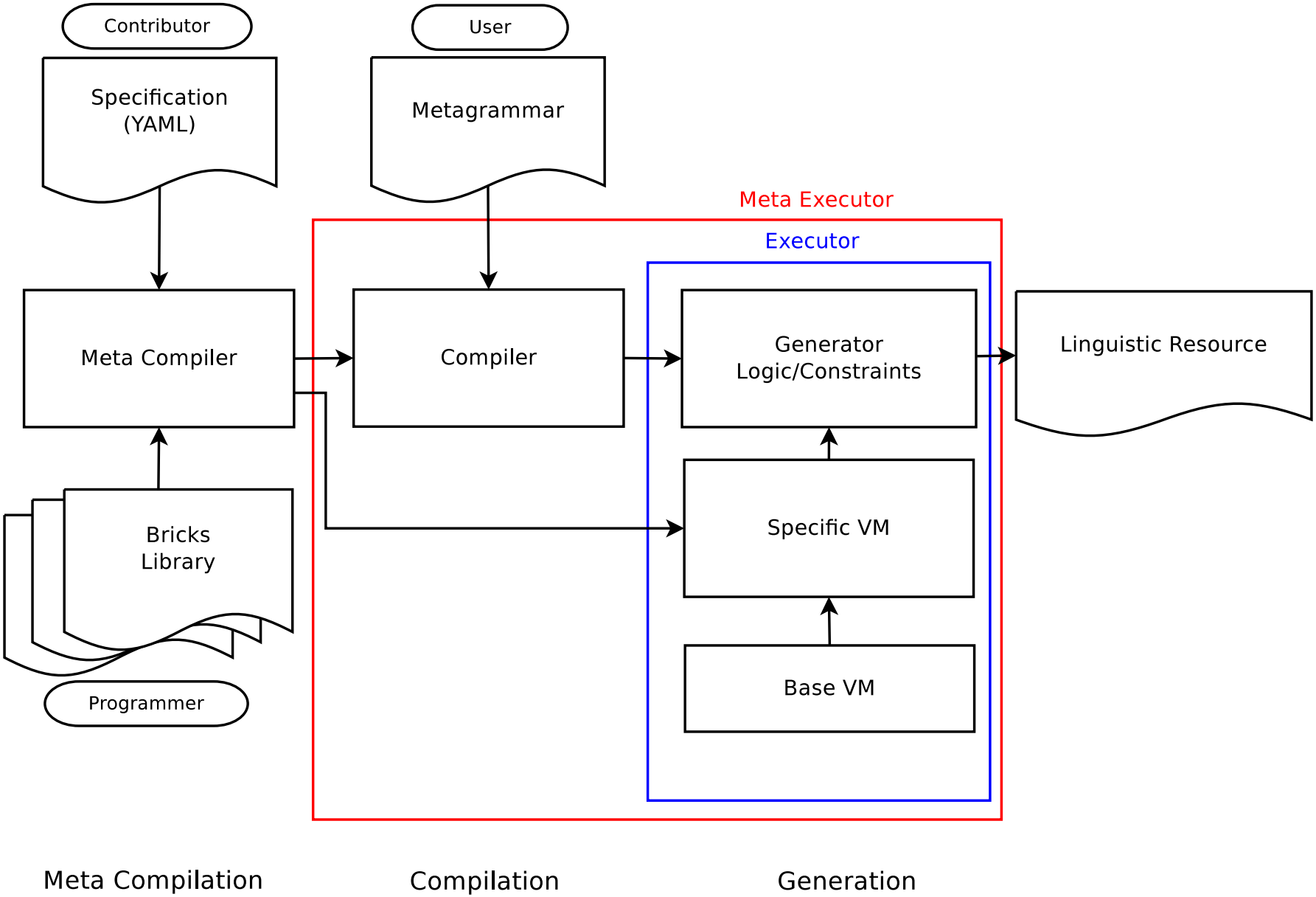
XMG-1

- eXtensible (?) Metagrammar
- Only 3 dimensions

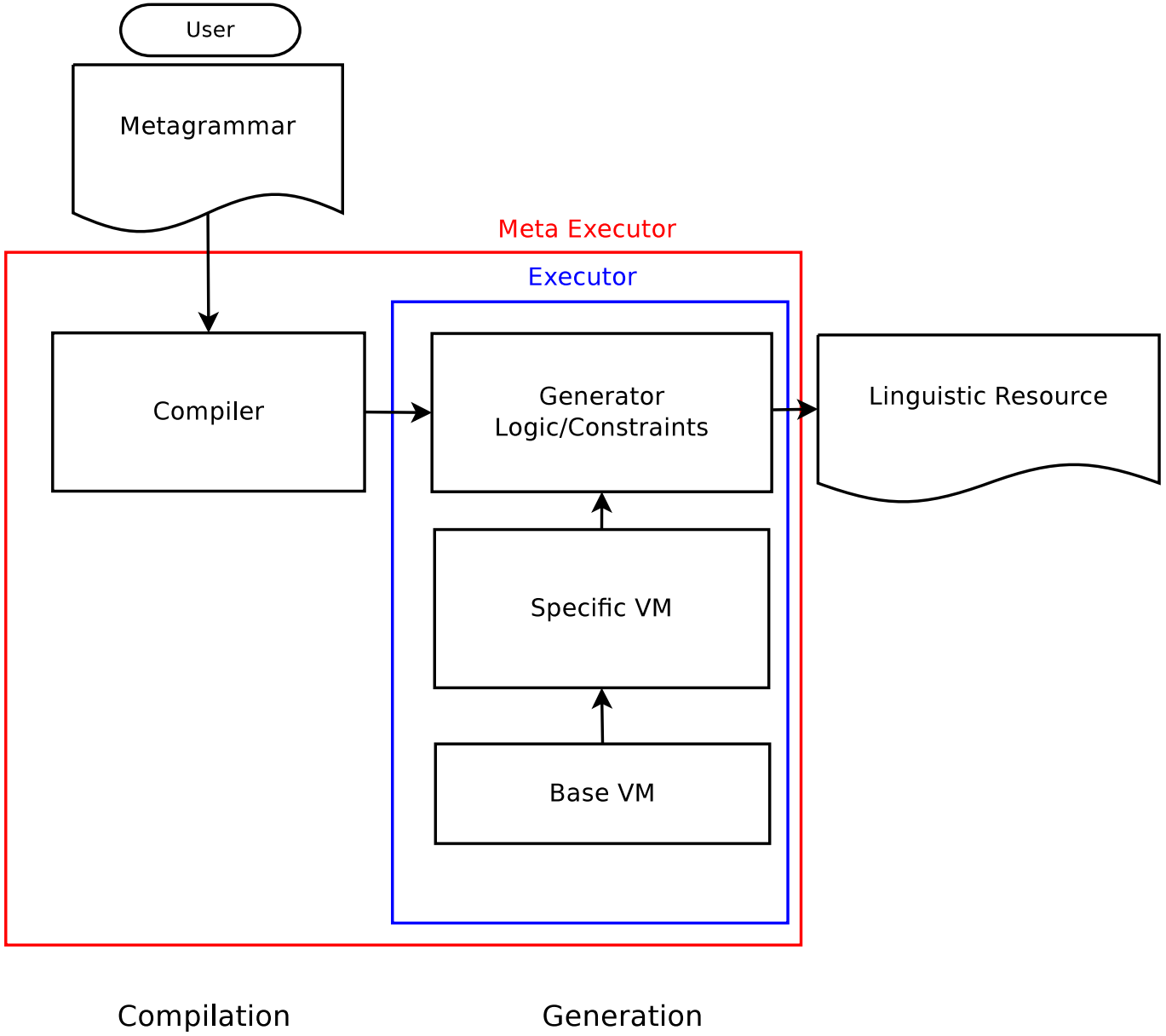
XMG-2

- Arbitrarily many dimensions, with DSLs
- Modular assembly of DSL, using bricks
- Methodology to generate a whole processing chain

XMG-2: Architecture



XMG-2: Architecture (relevant part for us)



Installing XMG 2

Three options, provided by the documentation:
`dokufarm.phil.hhu.de/xmg`

- Follow the steps (Ubuntu), or
- Install VirtualBox and get the XMG image
- Use the online compiler(s): `http://xmg.phil.hhu.de/index.php/upload/compile_grammar`

Installing contributions

- XMG bricks are distributed as contributions
- Making a contribution available is done with the `install` command

```
xmg@xmg:~/xmg-ng$ cd contributions
xmg@xmg:~/xmg-ng/contributions$ xmg install core
xmg@xmg:~/xmg-ng/contributions$ xmg install treemg
xmg@xmg:~/xmg-ng/contributions$ xmg install compat
xmg@xmg:~/xmg-ng/contributions$ xmg install
    synsemCompiler
```

Installing compilers

- A set of already assembled compilers is available
- Building one of them can be done with the `build` command

```
xmg@xmg:~/xmg-ng$ cd contributions/synsemCompiler/
xmg@xmg:~/xmg-ng/.../synsemCompiler$ cd compilers/
    synsem/
xmg@xmg:~/xmg-ng/.../synsem$ xmg build
```

- To avoid these steps: scripts (`reinstall.sh`)

Compiling a first metagrammar

The `compile` command takes two arguments

- The compiler which will be used
- The metagrammar

```
xmg@xmg:~/xmg-ng$ xmg compile synsem MetaGrammars/synsem
/TagExample.mg
```

Drawing trees

The output of XMG2 can be given to a parser or a generator, but also be inspected by a tree viewer

- XMG comes with a built-in tree viewer:

```
xmg@xmg:~/xmg-ng$ xmg gui tag
```

- Pytreeview (`https://gitlab.com/parmenti/pytreeview`) is a light tree viewer installed on the Virtualbox distribution of XMG2:

```
xmg@xmg:~/xmg-ng$ pytreeview --mode WEB -i input-file.
    xml
```

- A tree and frame viewer is available online: `http://xmg.phil.hhu.de/index.php/upload/xmg_viewer`

The control language

XMG descriptions:

- Associate a content to an identifier (abstraction)
- Describe structures inside dimensions, with dedicated languages
- Use other abstractions (classes)
- Combine contents in a disjunctive or a conjunctive way

$$
\begin{aligned}
\mathit{Class} &:= \quad \mathit{Name} \longrightarrow \mathit{Content} \\
\mathit{Content} &:= \quad \langle \mathit{Dimension} \rangle \{ \mathit{Description} \} \mid \mathit{Name} \mid \\
&\qquad\quad \mathit{Content} \vee \mathit{Content} \mid \mathit{Content} \wedge \mathit{Content}
\end{aligned}
$$

Describing trees

The <syn> dimension

- Declaring nodes: keyword **node**, optional node variable, optional features and properties

  ```
  node ?S [cat=s]
  ```

- Expressing constraints between nodes: dominance operators (`->`, `->+`, `->*`) and precedence operators (`>>`, `>>+`, `>>*`)
- Combining these statements: with logical operators (`;` and `|`)

Example:

```
1      node ?S [cat=s];
2      node ?VP [cat=vp];
3      node ?V (mark=anchor) [cat=v];
4      node ?NP (mark=subst) [cat=n];
5      ?S -> ?VP;
6      ?VP -> ?V;
7      ?S -> ?NP;
8      ?NP >> ?VP
```

Alternative syntax: bracket notation

The **<syn>** dimension

- Declaring nodes: same as for the standard notation
- Expressing dominance and precedence constraints thanks to bracketing, and special operators for non immediate relations (`...` , `...+` , `,,,` , `,,,+`)

```
1     node ?S [cat=s]{
2       node ?NP (mark=subst) [cat=np]
3       node ?VP [cat=vp]{
4         node ?V (mark=anchor) [cat=v]
5       }
6     }
```

Using dimensions

Contributing descriptions

- Descriptions (constraints) are accumulated into dimensions
- Every dimension is associated to a solver (sometimes identity)
- **<syn>**: a tree solver generates all minimal models

```
1  <syn>{
2      node ?S [cat=s];
3      node ?VP [cat=vp];
4      node ?V (mark=anchor) [cat=v];
5      node ?NP (mark=subst) [cat=n];
6      ?S -> ?VP;
7      ?VP -> ?V;
8      ?S -> ?NP;
9      ?NP >> ?VP
10 }
```

Syntactic nodes

Two nodes can be unified if:

- their feature structures can be unified
- their properties can be unified

Unification of nodes happens at two different stages:

- During the execution of the code ("explicit" unification: unification instruction **=** or reuse of variable)
- After solving: some nodes may be merged to obtain a minimal model

Minimal models

A minimal model is a model of the description where:

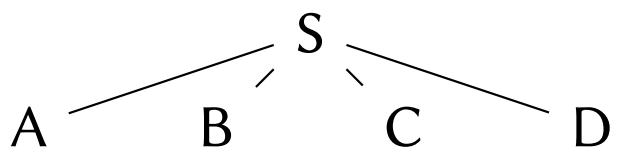
- no constraint is violated
- no additional node is created

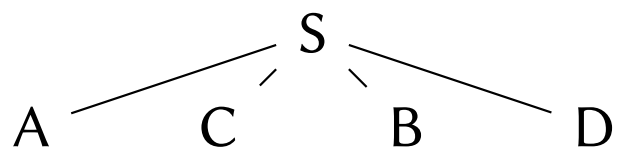
What are the minimal models for the following sets of constraints?

```
1  ?S ->+ ?A ; ?S -> ?B
```

```
1  ?S -> ?A ; ?S -> ?B ; ?S -> ?C ; ?A >>* ?C
```

Which set of constraints leads to the following minimal models?





Defining abstractions

Classes allow to:

- Control the scope of variables
- Make (parametrized) abstractions

Examples (just headers):

```
1 class kicked_the_bucket
2 import nx0Vnx1[]
3 declare ?X0 ?X1
```

```
1 class nx0Vnx1
2 export ?S ?NP_Subj ?VP ?V ?NP_Obj
3 declare ?S ?NP_Subj ?VP ?V ?NP_Obj ?X0 ?X1
```

Defining abstractions

```
1  class Intransitive
2  declare ?S ?NP ?VP ?V
3  {
4    <syn>{
5        node ?S [cat=s];
6        node ?VP [cat=vp];
7        node ?V (mark=anchor) [cat=v];
8        node ?NP (mark=subst) [cat=n];
9        ?S -> ?VP; ?VP -> ?V;
10       ?S -> ?NP; ?NP >> ?VP
11   }
12 }
```

Valuation

To specify for which class models have to be computed (the axioms), the instruction **value** has to be used after the class definitions.

```
1  value Intransitive
2  value Transitive
```

Using abstractions

Classes can be used by other classes by two means:

- Importing the class in the header: all the (exported) variables are added to the scope, all the constraints from the class are added to the current set of constraints
- Calling the class in the body: variables are not added to the scope

Calling classes has two advantages:

- alternatives are possible (disjunction)
- it allows to use parameters

Examples:

```
1 CanObj[] | RelObj[]
```

```
1 ?C=Class[?X]
```

Classes: examples (1)

```
1  class a
2  export ?A
3  declare ?A ?S
4  {
5    <syn>{
6      ?S -> ?A
7    }
8  }
9
10 class b
11 import a[]
12 declare ?B
13 {
14   <syn>{
15     ?B -> ?A
16   }
17 }
```

Classes: examples (2)

```
1  class a
2  export ?S
3  declare ?A ?S
4  {
5    <syn>{
6      ?S -> ?A
7    }
8  }
9
10 class b
11 import a[]
12 declare ?A
13 {
14   <syn>{
15     ?S -> ?A
16   }
17 }
```

Definition of types and constants

Everything inside the metagrammar has a type: values, feature structures, nodes, dimensions…

Four ways to define new types:

- Enumerated type: type T={a,b,c,d}
- Structured type: type T=[a_1:t_1,…,a_n:t_n]
- Interval type: type T=[1..3]
- Unspecified type: type T!

Definition of types and constants

We can now specify the types of features and properties:

```
1  type CAT=  {np,vp,s,n,v,det}
2  type MARK= {lex,anchor,subst}
3  type LABEL !
4  type PERS= [1..3]
5  type GEN = {m,f}
6  type NUM = {sg,pl}
7  type AGR = [gen:GEN, num:NUM]
8
9
10 feature cat:  CAT
11 feature e:    LABEL
12 feature pers: PERS
13 feature agr:  AGR
14
15 property mark: MARK
```

Outline

Principles: motivation

- As fragments become more numerous, controlling their combination (and the scope of variables) gets difficult
- Idea: adding new constraints on top of dominance and precedence
- Principles: sets of additionnal constraints for the solver[CrabbeDuchier:04]

A set of principles

XMG offers several sets of additionnal constraints over the models (principles):

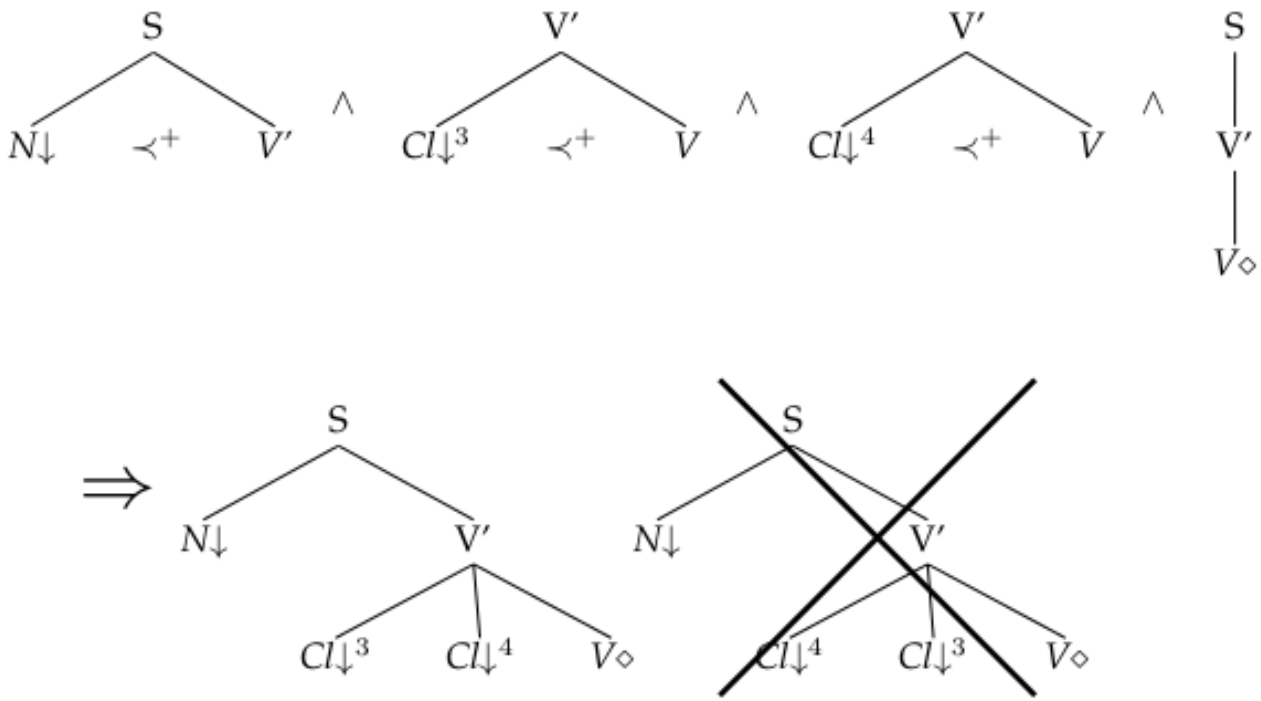
- colors: polarities for node unification
- rank: linear order constraints on nodes
- unicity: uniqueness of a feature inside a model

Rank: Clitics ordering

- The ordering of clitic pronouns (in Spanish or French for example) is known to be problematic when formalizing a grammar
- In a metagrammar, when combining fragments, nodes representing these clitics have to come in a specific order

- Pedro nos la da
- *Pedro la nos da
- Je le lui laisse
- *Je lui le laisse

Rank: Clitics ordering (in French)



- Every produced model has to satisfy the order constraint

Using principles: rank

```
use rank with () dims (syn)
type RANK=[1..7]
property rank: RANK
```

```
class CliticIobjectII
import nonReflexiveClitic[]
{
  <syn>{
    node xCl(rank=2)
            [top=[func=iobj, pers = @{1,2}]]
  }
}
```

Using principles: unicity

```
1 use unicity with (rank=1) dims (syn)
2 use unicity with (rank=2) dims (syn)
3 use unicity with (rank=3) dims (syn)
4 use unicity with (rank=4) dims (syn)
5 use unicity with (rank=5) dims (syn)
6 use unicity with (rank=6) dims (syn)
7 use unicity with (rank=7) dims (syn)
```

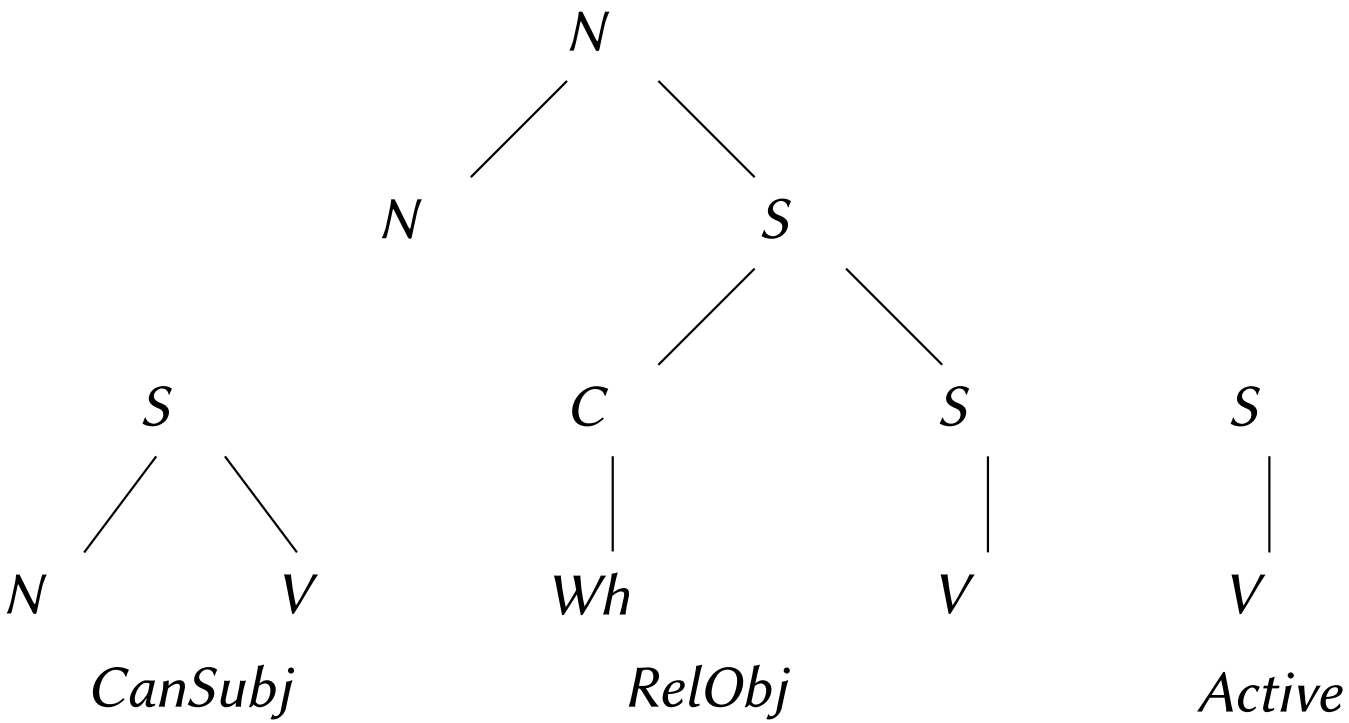
Using principles: colors

- Colors are a solution to guide the combination of fragments
- A color is affected to every node
- New constraints on node unification

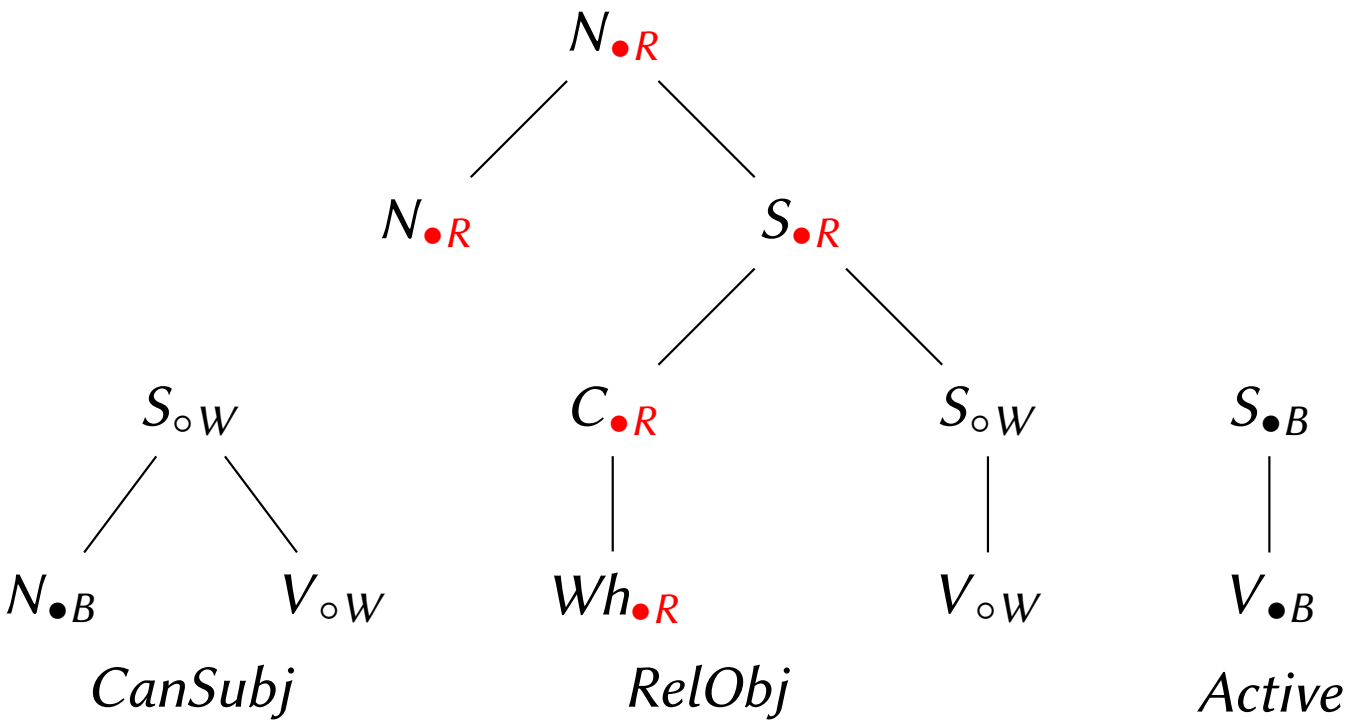
	$\bullet_B$	$\bullet_R$	$\circ_W$	$\bot$
$\bullet_B$	$\bot$	$\bot$	$\bullet_B$	$\bot$
$\bullet_R$	$\bot$	$\bot$	$\bot$	$\bot$
$\circ_W$	$\bullet_B$	$\bot$	$\circ_W$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

- Valid models only have red and black nodes

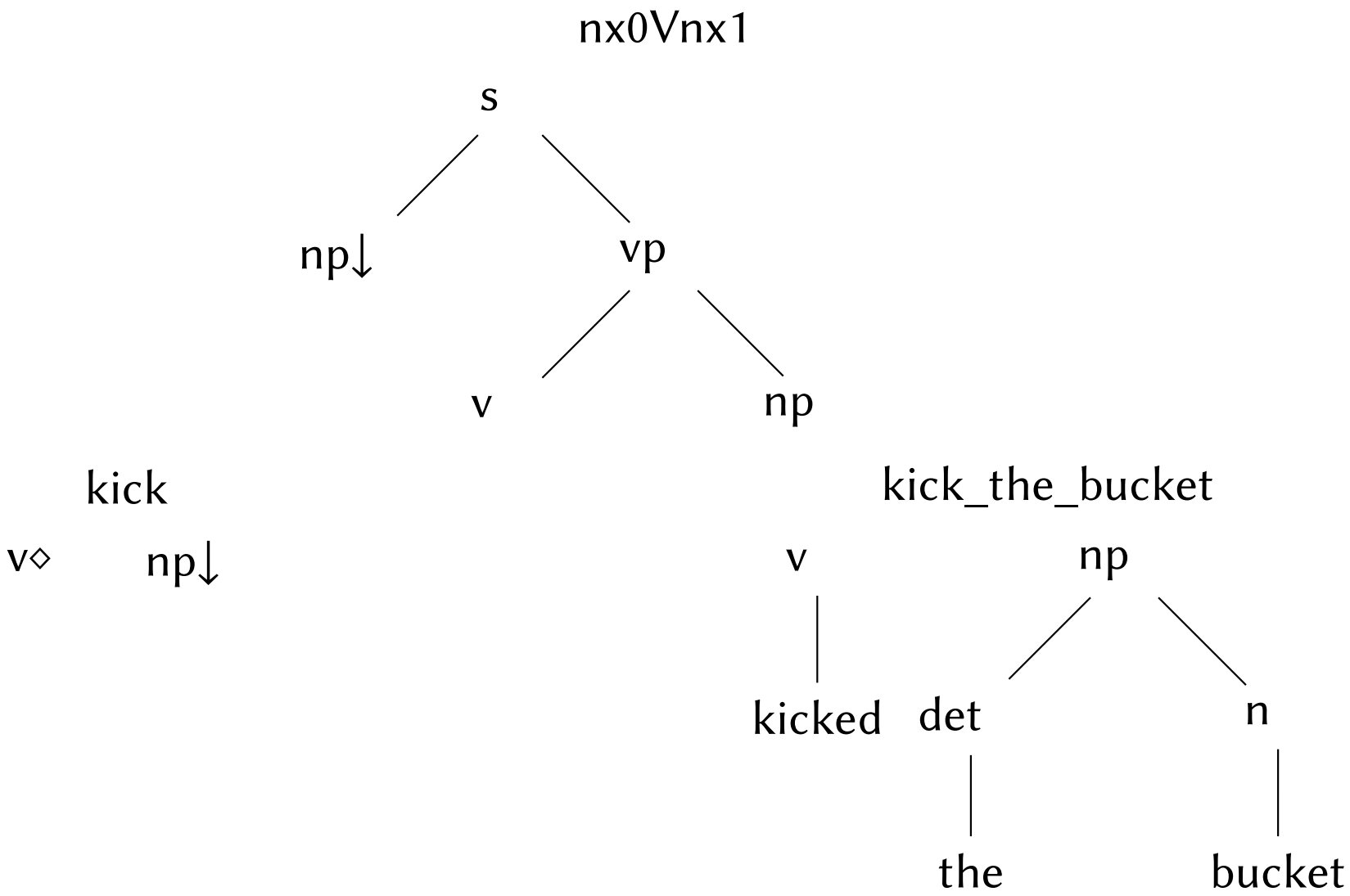
Combination with polarities



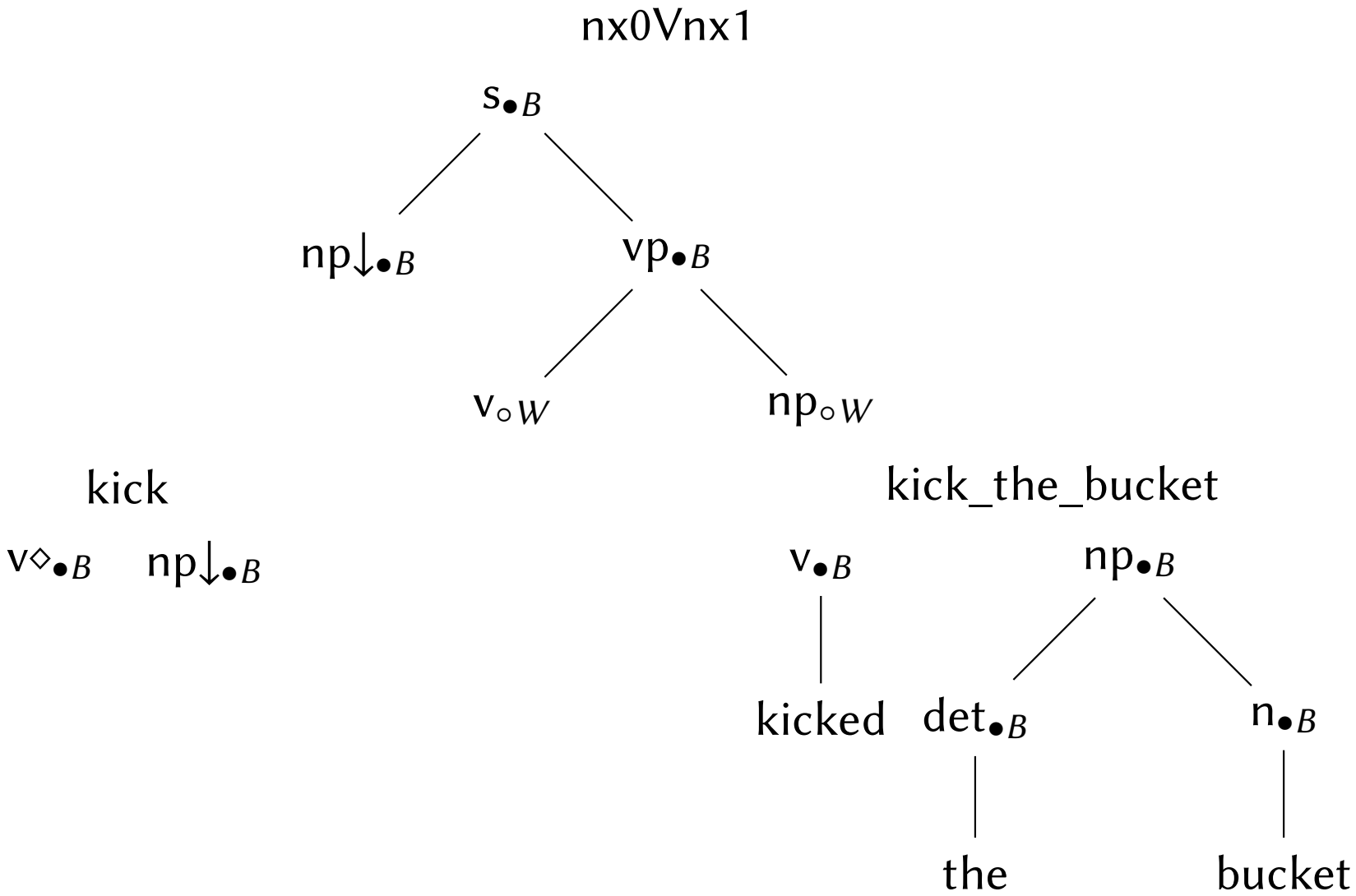
Combination with polarities



Combination with polarities



Combination with polarities



Using principles: colors

```
use color with () dims (syn)
type COLOR={red,black,white}
property color: COLOR
```

```
class nx0Vnx1
declare ?S ?NP_Subj ?VP ?V ?NP_Obj
{
  <syn>{
     ?S (color=red)[cat=s] {
       ?NP_Subj (color=black, mark=subst) [cat=np]
       ?VP (color=black)[cat=vp] {
         ?V (color=white)[cat=v]
         ?NP_Obj (color=white)[cat=np]
       }
     }
  }
}
```

Outline

Summary

- A metagrammar contains descriptions of unanchored elementary trees.
- Metagrammar descriptions are declarative and multidimensional.
- Metagrammar descriptions make up an inheritance hierarchy.
- The metagrammar allows one to express and implement lexical generalizations, e.g. active-passive diathesis.

Summary

- A metagrammar contains descriptions of unanchored elementary trees.
- Metagrammar descriptions are declarative and multidimensional.
- Metagrammar descriptions make up an inheritance hierarchy.
- The metagrammar allows one to express and implement lexical generalizations, e.g. active-passive diathesis.

Hot topics:

- parsing with metagrammars[7]
- use metagrammars for morphological descriptions[11,18]

Summary

- A metagrammar contains descriptions of unanchored elementary trees.
- Metagrammar descriptions are declarative and multidimensional.
- Metagrammar descriptions make up an inheritance hierarchy.
- The metagrammar allows one to express and implement lexical generalizations, e.g. active-passive diathesis.

Hot topics:

- parsing with metagrammars[7]
- use metagrammars for morphological descriptions[11,18]

Adjacent topics:

- grammar induction from treebanks[5,6,13,22]

[1] Alahverdzhieva, Katya. 2008. *XTAG using XMG. A core Tree-Adjoining Grammar for English*. University of Nancy 2 / University of Saarland Master's Thesis. http://homepages.inf.ed.ac.uk/s0896251/pubs/msc-sb2008.pdf.

[2] Becker, Tilman. 1994. *Hytag: a new type of tree adjoining grammars for hybrid syntactic representations of free word order languages*. Universität des Saarlandes dissertation. http://www.dfki.de/~becker/becker.diss.ps.gz.

[3] Becker, Tilman. 2000. Patterns in metarules for TAG. In Anne Abeillé & Owen Rambow (eds.), *Tree Adjoining Grammars: Formalisms, linguistic analyses and processing* (CSLI Lecture Notes 107), 331–342. Stanford, CA: CSLI Publications.

[4] Candito, Marie-Hélène. 1996. A principle-based hierarchical representation of LTAGs. In *Proceedings of the 16th international Conference on Computational Linguistics (COLING 96)*. Copenhagen. http://aclweb.org/anthology-new/C/C96/C96-1034.pdf.

[5] Chen, John, Srinivas Bangalore & K. Vijay-Shanker. 2006. Automated extraction of Tree-Adjoining Grammars from treebanks. *Natural Language Engineering* 12. 251–299.

[6] Chiang, David. 2000. Statistical parsing with an automatically-extracted Tree Adjoining Grammar. In *Proceedings of the 38th annual meeting of the Association for Computational Linguistics*, 456–463. Hong Kong.

[7] de la Clergerie, Éric Villemonte. 2013. Exploring beam-based shift-reduce dependency parsing with DyALog: results from the SPMRL 2013 shared task. In *4th workshop on statistical parsing of morphologically rich languages (SPMRL'2013)*. Seattle. http://hal.inria.fr/docs/00/87/91/29/PDF/dyalogsr.pdf.

[8] Crabbé, Benoît. 2005. *Représentation informatique de grammaires d'arbres fortement lexicalisées: Le cas de la grammaire d'arbres adjoints*. Université Nancy 2 dissertation.

[9] Crabbé, Benoit, Denys Duchier, Claire Gardent, Joseph Le Roux & Yannick Parmentier. 2013. XMG: eXtensible MetaGrammar. *Computational Linguistics* 39(3). 1–66. http://hal.archives-ouvertes.fr/hal-00768224/en/.

[10] Dowty, David R. 1979. *Word meaning and Montague Grammar*. Reprinted 1991 by Kluwer Academic Publishers. Dordrecht: D. Reidel Publishing Company.

[11] Duchier, Denys, Brunelle Magnana Ekoukou, Yannick Parmentier, Simon Petitjean & Emmanuel Schang. 2012. Describing morphologically rich languages using metagrammars: A look at verbs in Ikota. In *Workshop on language technology for normalisation of less-resourced languages (SALTMIL 8 – AfLaT 2012)*, 55–59. http://www.tshwanedje.com/publications/SaLTMiL8-AfLaT2012.pdf#page=67.

[12] Gazdar, Gerald. 1981. Unbounded dependencies and coordinated structure. *Linguistic Inquiry* 12. 155–182.

[13] Kaeshammer, Miriam & Vera Demberg. 2012. German and English treebanks and lexica for Tree-Adjoining Grammars. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk & Stelios Piperidis (eds.), *Proceedings of the eighth international Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey: European Language Resources Association (ELRA).

[14] Kallmeyer, Laura, Timm Lichte, Wolfgang Maier, Yannick Parmentier & Johannes Dellert. 2008. Developing a TT-MCTAG for German with an RCG-based parser. In European Language Resources Association (ELRA) (ed.), *Proceedings of the sixth international Conference on Language Resources and Evaluation (LREC'08)*. Marrakech, Morocco.

[15] Lichte, Timm & Simon Petitjean. 2015. Implementing semantic frames as typed feature structures with XMG. *Journal of Language Modelling* 3(1). 185–228. http://jlm.ipipan.waw.pl/index.php/JLM/article/view/96.

[16] Parmentier, Yannick, Laura Kallmeyer, Wolfgang Maier, Timm Lichte & Johannes Dellert. 2008. TuLiPA: A syntax-semantics parsing environment for mildly context-sensitive formalisms. In *Proceedings of the ninth international workshop on Tree Adjoining Grammars and related formalisms (TAG+9)*, 121–128. Tübingen, Germany.

[17] Petitjean, Simon. 2014. *Génération Modulaire de Grammaires Formelles*. Orléans, France: Université d'Orléans Thèse de Doctorat. https://tel.archives-ouvertes.fr/tel-01163150/.

[18] Petitjean, Simon, Younes Samih & Timm Lichte. 2015. Une métagrammaire de l'interface morpho-sémantique dans les verbes en arabe. In *Actes de la 22e conférence sur le Traitement Automatique des Langues Naturelles*, 473–479. Caen, France. http://www.atala.org/taln_archives/TALN/TALN-2015/taln-2015-court-024.

[19] Prolo, Carlos A. 2002. Generating the XTAG English grammar using metarules. In *Proceedings of the 19th international Conference on Computational Linguistics (COLING 2002)*, 814–820. Taipei. Taiwan.

[20] Ristad, Eric Sven. 1987. Revised General Phrase Structure Grammar. In *Proceedings of the 25th annual meeting of the Association for Computational Linguistics*, 243–250. Stanford, CA. http://www.aclweb.org/anthology/P87-1034.

[21] Uszkoreit, Hans & Stanley Peters. 1987. On some formal properties of metarules. English. In Walter J. Savitch, Emmon Bach, William Marsh & Gila Safran-Naveh (eds.), *The formal complexity of natural language* (Studies in Linguistics and Philosophy 33), 227–250. Dordrecht, The Netherlands: D. Reidel Publishing. http://dx.doi.org/10.1007/978-94-009-3401-6_9.

[22] Xia, Fei. 2001. *Automatic grammar generation from two different perspectives*. University of Pennsylvania dissertation.
`http://faculty.washington.edu/fxia/papers_from_penn/thesis.pdf`.
[23] XTAG Research Group. 2001. *A Lexicalized Tree Adjoining Grammar for English*.
Tech. rep. Philadelphia, PA: Institute for Research in Cognitive Science, University of Pennsylvania.