# Grammar Implementation with TAG

## XMG - eXtended MetaGrammar

Timm Lichte

HHU Düsseldorf

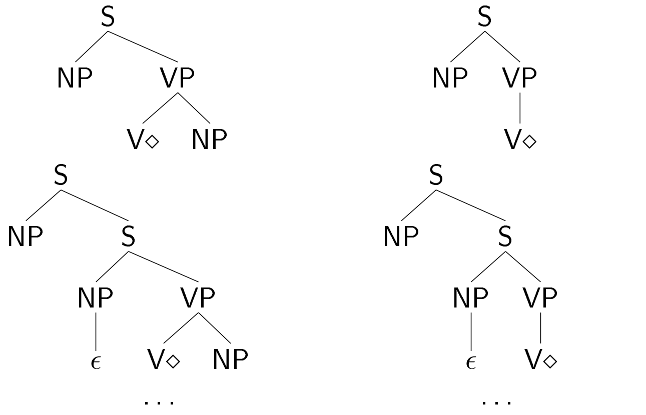
SS 2011

01.06.2011

## The situation

**39 templates for transitive verbs**

**12 tree templates for intransitive verbs**



Basically, XTAG defines a set of 221 unrelated tree templates.

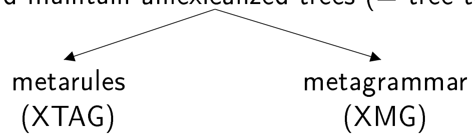
## A task for grammar engineering

**General task**

Generate and maintain a large-coverage LTAG!

**Subtasks:**

1. Generate and maintain unlexicalized trees (= tree templates)!



metarules (XTAG)

metagrammar (XMG)

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

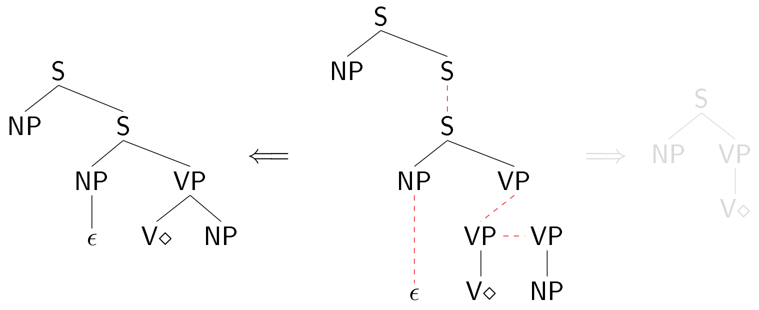
## Metagrammars

- additional layer of abstraction at the level of tree templates

⇒ allow for the description of **tree fragments**

- A tree template is the combination of tree fragments.

⇒ Tree fragments can be reused!



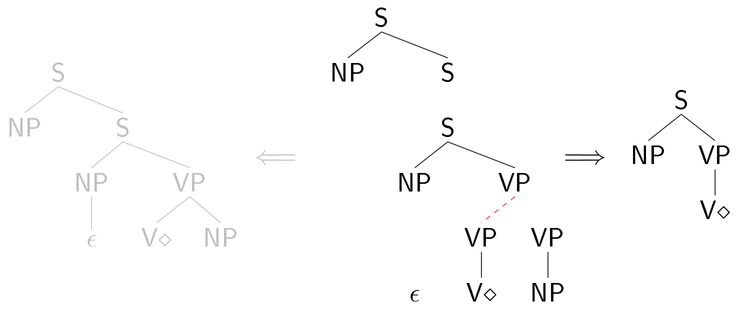
# Metagrammars

- additional layer of abstraction at the level of tree templates

⇒ allow for the description of **tree fragments**

- A tree template is the combination of tree fragments.

⇒ Tree fragments can be reused!



# XMG - Background

- name of the metagrammar formalism and of a metagrammar compiler
- developed at LORIA, Nancy, France
- written in Oz/Mozart
- available at `http://sourcesup.cru.fr/xmg`

⇒ Other metagrammar implementations exist, but XMG is the most elaborate one.

# XMG - Description languages

**$\mathcal{L}_D$: Description language for tree fragments**

Let ?x and ?y be nodes:

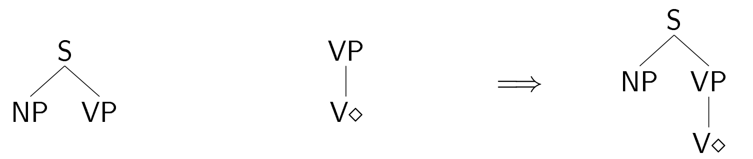
```
                 ( ?x -> ?y | ?x ->+ ?y | ?x ->* ?y |   )
                 ( ?x ≫ ?y | ?x ≫+ ?y | ?x ≫* ?y |      )
Description ::=  ( ?x = ?y |                            )
                 ( ?x[f:E] | ?x(p:E) |                  )
                 ( Description ∧ Description            )
```

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

```
Class ::= Name ⟶ Content

             ( Description | Name |  )
Content ::=  ( Content ∨ Content |   )
             ( Content ∧ Content     )
```

# XMG - Description languages - Examples



```
{ ?S -> ?NP,         { ?VP2 -> ?V,          { ?S -> ?NP,
  ?S -> ?VP1,          ?VP2[cat:vp],          ?S -> ?VP1,
  ?NP ≫ ?VP1           ?V[cat:v],             ?NP ≫ ?VP1
  ?S[cat:s],           ?V(mark:anchor) }      ?S[cat:s],
  ?NP[cat:np],                                ?NP[cat:np],
  ?VP[cat:vp] }                               ?VP[cat:vp],
                                              ?VP2 -> ?V,
                                              ?VP2[cat:vp],
                                              ?V[cat:v],
                                              ?V(mark:anchor) }

  Subject      ∧     VerbProjection    ⟵      intransitive
```

# XMG - Node variables and compiling

- Node variables have a scope local to the class (= name space).
- Tree descriptions can denote more than one tree fragment! BUT: Each of the tree fragments has to comply with all of the tree descriptions!

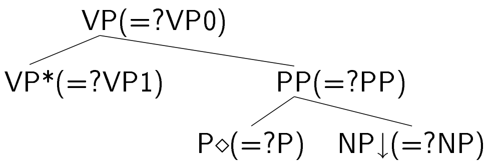
When the class `intransitive` is compiled:

1. XMG accumulates all tree descriptions involved in `intransitive`, and
2. XMG identifies tree fragments and tree templates by merging node variables or drawing edges.

In the previous example, the node variables ?VP1 and ?VP2 can be merged.

# XMG - The source code - The structure of trees

There are two ways to encode the structure of trees: (1) through tree descriptions, or (2) through brackets and linear order.



```
class betavxPnx
declare ?VP0 ?VP1 ?PP ?P ?NP
{<syn>{
node ?VP0; node ?VP1;
node ?PP; node ?NP;
node ?P;
?VP0 -> ?VP1; ?VP0 -> ?PP;
?PP -> ?P; ?PP -> ?NP;
?VP1 >> ?PP; ?P >> ?NP
}}
```

```
class betavxPnx
declare ?VP0 ?VP1 ?PP ?P ?NP
{<syn>{
node ?VP0 {
   node ?VP1
   node ?PP {
     node ?P
     node ?NP
   }
} }}
```

# XMG - The source code - Properties and feature structures

Firstly, the value types of features and properties have to be declared.

```
type MARK = {subst, foot, anchor, coanchor, flex }
type CAT = {np,v,vp,s}
```

Secondly, properties and features must be declared as well.

```
property mark : MARK
feature cat : CAT
```

Finally, properties and features of nodes can be specified.

```
class betavxPnx
{ ...
node ?NP (mark = subst) [cat = np]
... }
```

# XMG - The source code - Complex feature structures

**How to declare and use complex features?**

```
type ARG = [  3rdsing : bool,
              num : NUM,
              pers : PERS,
              gen : GEN        ]
feature arg:ARG
...
node ?NP [arg = [3rdsing = +] ]
...
```

**Top-bottom-feature-structures**

In XMG, there are predefined complex features top and bot for the specification of top-bottom-feature structures. Otherwise, feature specifications hold for both top and bottom.

**Note:** Links between features can be established by variables!

# XMG - The source code - Reusing classes

**General convention:** Names of reused classes have [] as a postfix.

### First method:

Class instantiations can be assigned to variables in the body. Only exported variables of the class can be used by means of the dot operator.

```
class betavxPnx
{ ...
?VPSpine = VPSpine[];
?VPSpine.?VP0 = ?XP;
... }
```

### Second method:

Classes can be imported, such that all variables of the imported class, that have been exported, can be used directly.

```
class betavxPnx
import VPSpine[]
{...
?VP0 = ?XP;
... }
```