Tree Adjoining Grammars

Grammar Implementation with XMG

Laura Kallmeyer & Timm Lichte

HHU Düsseldorf

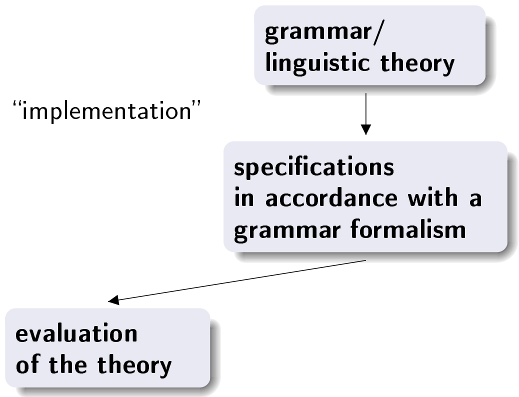
WS 2012

03.12.2012

Outline

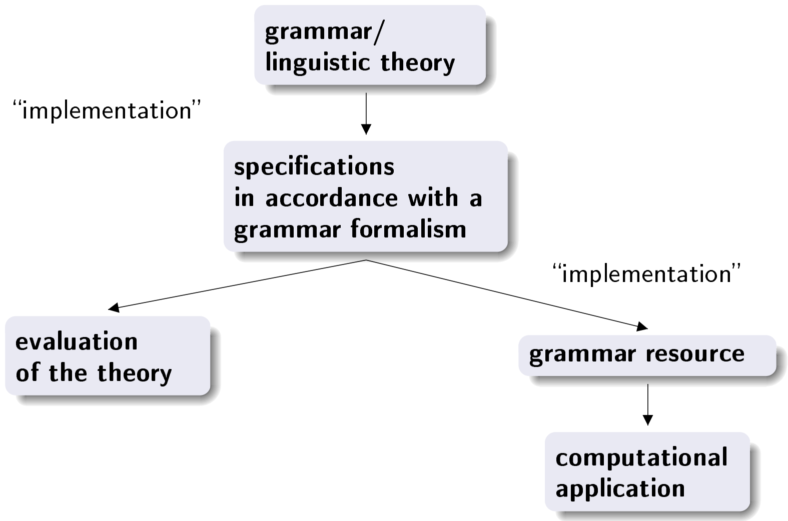
1. What is grammar implementation?
2. Two ways of tree template implementation:
   - Metarules
   - Metagrammars
3. eXtended Metagrammar (XMG)
4. A case study with XMG

Two kinds of grammar implementation

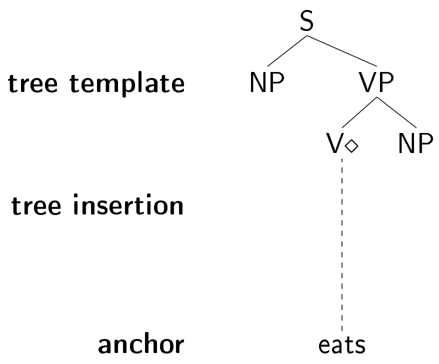


As is frequently pointed out but cannot be overemphasized, an important goal of formalization in linguistics is to enable subsequent researchers ***to see the defects of an analysis as clearly as its merits****; only then can progress be made efficiently.* [Dowty, 1979, 322]

Two kinds of grammar implementation



What kind of grammar resource?



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

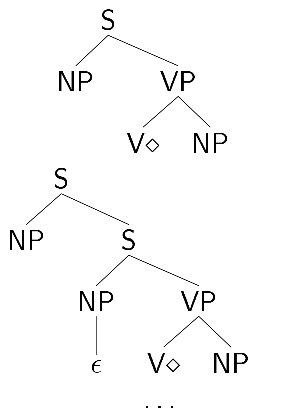
- **XTAG tools** [XTAG Research Group, 2001]
  1. implementation tools (with metarules)
  2. editor/viewer for MorphDB and SynDB
  3. parser
- **XMG + lexConverter + TuLiPA**
  1. XMG: eXtensible MetaGrammar [Duchier et al., 2004]
  2. lexConverter (LEX2ALL)
  3. TuLiPA: Tübingen Linguistic Parsing Architecture [Parmentier et al., 2008]

Outline

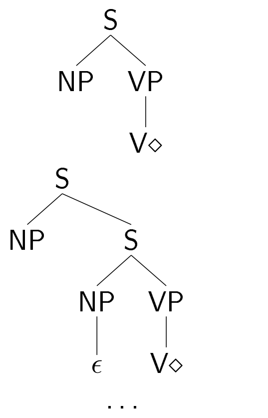
1. What is grammar implementation?
2. Two ways of tree template implementation:
   - Metarules
   - Metagrammars
3. eXtended Metagrammar (XMG)
4. A case study with XMG

The situation

39 templates for transitive verbs



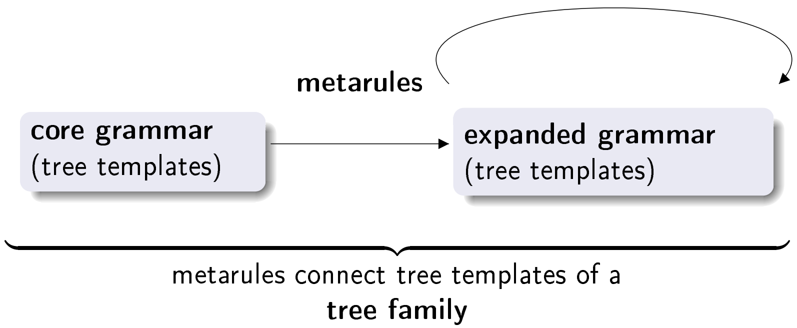
12 tree templates for intransitive verbs



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

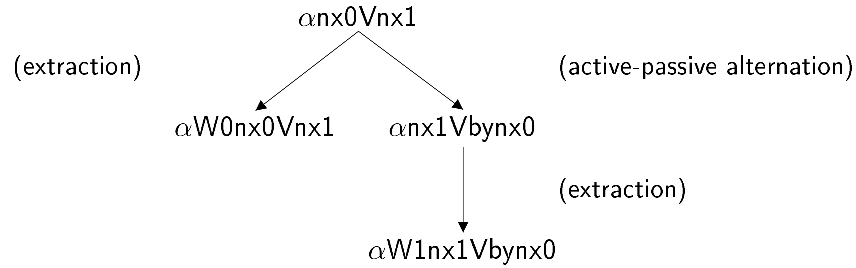
Metarules for LTAG

[Becker, 1994], [Becker, 2000], [Prolo, 2002]
Idea from GPSG [Gazdar, 1981]



Metarules for LTAG: Example

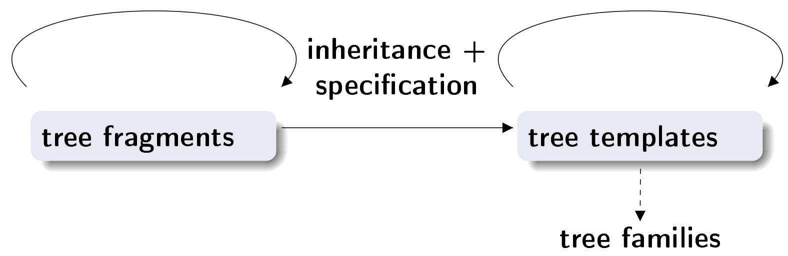
Tnx0nx1:



Metarules do not only add structure, they can also eliminate structure!

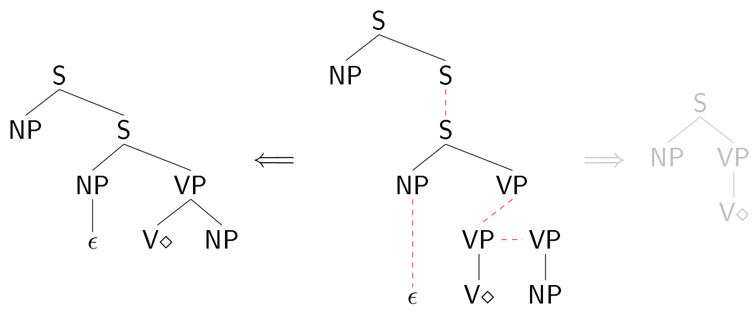
Metagrammars for LTAG

[Candito, 1996], [Xia, 2001], [Crabbé, 2005]

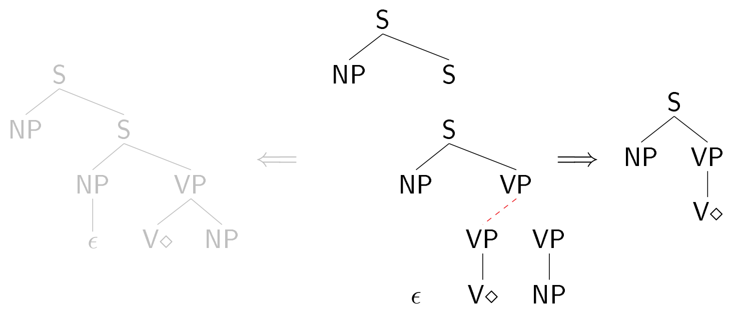


- **tree fragments**: additional layer of abstraction below the level of tree templates
- A tree template is the result of combining and specifying tree fragments and tree templates.
- The notion of **tree families** is independent from the construction of tree templates!

Metagrammars for LTAG: Example



Metagrammars for LTAG: Example



Outline

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.

Some existing implementations using XMG:

- French: FrenchTAG [Crabbé, 2005]
- English: XTAG with XMG [Alahverdzhieva, 2008]
- German: GerTT [Kallmeyer et al., 2008]

XMG - Description language for tree fragments

$\mathcal{L}_D$: 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           )
```

->	immediate dominance
->+	dominance (transitive, non-reflexive closure)
->*	reflexive dominance (transitive, reflexive closure)
>>	immediate precedence
>>+	precedence (transitive, non-reflexive closure)
>>*	reflexive precedence (transitive, reflexive closure)
?x[f=E]	feature declaration
?x(p=E)	property declaration

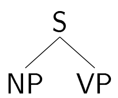
XMG - Description language for tree fragments

$\mathcal{L}_D$: 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           )
```

Example:



can be described as

```
{ ?S -> ?NP,
  ?S -> ?VP1,
  ?NP >> ?VP1,
  ?S[cat=s],
  ?NP[cat=np],
  ?VP1[cat=vp] }
```


XMG - Description language for tree fragments

Tree descriptions can denote more than one tree fragment!
BUT: Each of the tree fragments has to comply with all of the tree descriptions!

⇒ Infinitely many trees satisfy

```
{ ?S -> ?NP,
  ?S -> ?VP1,
  ?NP >> ?VP1,
  ?S[cat=s],
  ?NP[cat=np],
  ?VP1[cat=vp] }
```

XMG only considers the **minimal model** of a tree description, hence trees that only contain nodes given in the description.

XMG - Description language - Classes

Tree descriptions are encapsuled in so-called **classes**:

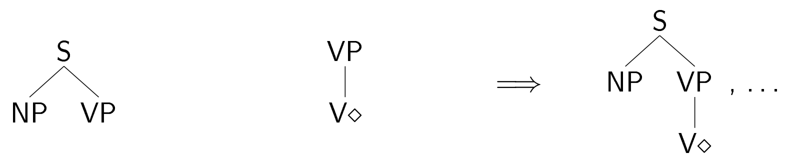
$\mathcal{L}_C$: Description language for the combination of tree descriptions

```
Class ::= Name → Content

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

- Node variables have a scope local to the class (= name space).
- When combining tree descriptions δ_1 and δ_2:
  1. XMG unifies δ_1 and δ_2, and
  2. XMG renames variables common variables.

XMG - Description languages - Examples




```
{ ?S -> ?NP,          { ?VP1 -> ?V,          { ?S -> ?NP,
  ?S -> ?VP1,           ?VP1[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],
  ?VP1[cat=vp] }                               ?VP1[cat=vp],
                                               ?VP2 -> ?V,
                                               ?VP2[cat=vp],
                                               ?V[cat=v],
                                               ?V(mark=anchor) }
```


XMG - The source code

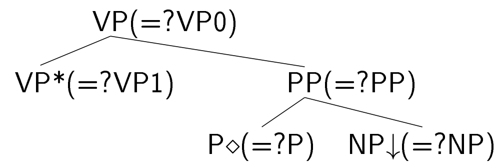
Tree fragments, tree templates and tree families are models of so-called **classes** (as known from object oriented programming).

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

- tree descriptions?
- feature structures?
- type of node (footnode, substitution node, anchor)?
- combination of trees?

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, nadj }
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 AGR = [   3rdsing : bool,
               num : NUM,
               pers : PERS,
               gen : GEN        ]
feature agr:AGR
...
node ?NP [agr = [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;
... }
```


XMG - The source code - Evaluation

Which class represents a tree template, i.e. which class needs to be evaluated by XMG?

⇒ This is expressed/triggered by the command **value**.

```
class betavxPnx
...
value betavxPnx
```


XMG - The source code - Putting the pieces together I

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

property mark : MARK
feature cat : CAT

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TREE FRAGMENTS:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class VPSpine
export ?VP0 ?VP1
declare ?VP0 ?VP1
{ <syn>{
    node ?VP0 [cat=vp]{
      node ?VP1 (mark=foot) [cat=vp]
  }
  }
}

class PrepositionalPhrase
export ?PP ?P ?NP
declare ?PP ?P ?NP
{ <syn>{
    node ?PP [cat=pp] {
      node ?P (mark=anchor) [cat=p]
      node ?NP (mark=subst) [cat=np]
    }
  }
}
```

XMG - The source code - Putting the pieces together II

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TREE TEMPLATES:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class betavxPnx
declare ?PrepP ?VPSpine
{
  ?PrepP = PrepositionalPhrase[];
  ?VPSpine = VPSpine[];
  <syn> {
    ?VPSpine.?VP0 -> ?PrepP.?PP;
    ?VPSpine.?VP1 >> ?PrepP.?PP
  }
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EVALUTATION:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

value betavxPnx
```


Outline

XMG - The source code - Declaring a tree family

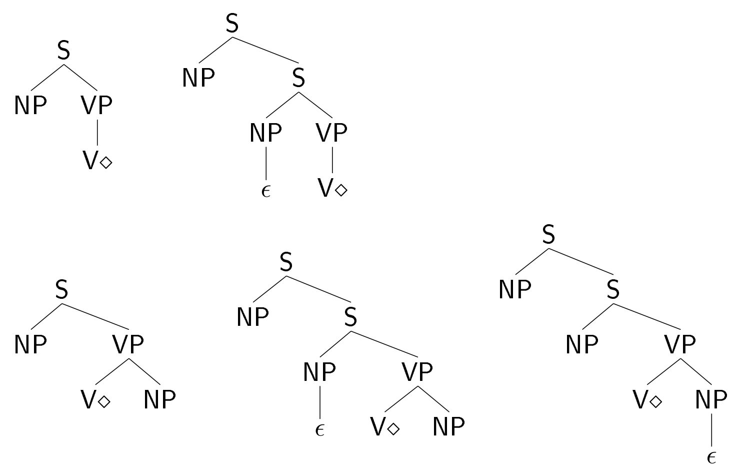
```
class Tnx0V
declare ?Tnx0V
{
        ?Tnx0V = ( alphanx0V[] | alphaW0nx0V[] )
}

...

value Tnx0V
```


XMG - Case study

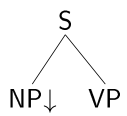
How to describe the tree families for intransitive (Tnx0V) and transitive (Tnx0Vnx1) tree templates?



XMG - Case study - The fragments

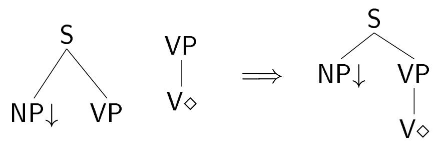

```
class VerbProjection
export ?VP ?V
declare ?VP ?V
{<syn>{
    node ?VP [cat = vp];
    node ?V (mark = anchor)[cat = v];
    ?VP -> ?V
  }
}
```


XMG - Case study - The fragments



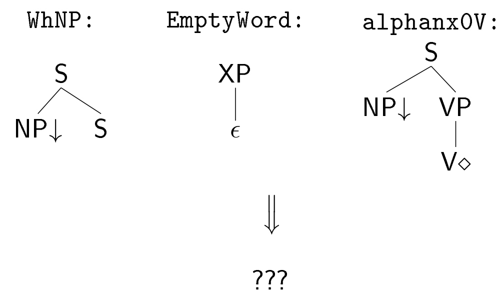

```
class Subject
export ?S ?NP ?VP
declare ?S ?NP ?VP
{ <syn>{
    node ?S [cat = s]{
        node ?NP (mark = subst)[cat = np]
        node ?VP [cat = vp]
        }
    }
}
```


XMG - Case study - Building a tree template




```
class alphanx0V
import VerbProjection[]
export ?S ?NP0
declare ?Subj ?S ?NP0
{
    ?Subj = Subject[];      ?NP0 = ?Subj.?NP;
    ?VP = ?Subj.?VP;        ?S = ?Subj.?S
}
```

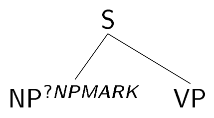

XMG - Case study - Adding fragments for extraction



In order to reuse `alphanx0V` here one has to underspecify the mark property of leaf nodes!

1. in subject and object fragments and in tree templates (e.g. `nx0V`)
2. only in subject and object fragments
3. ...

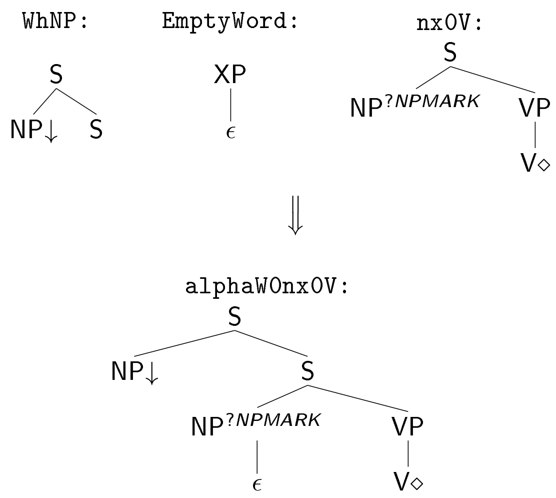
XMG - Case study - A redesigned subject fragment




```
class Subject
export ?S ?NP ?VP ?NPMARK
declare ?S ?NP ?VP ?NPMARK
{ <syn>{
    node ?S [cat = s]{
        node ?NP (mark = ?NPMARK)[cat = np]
        node ?VP [cat = vp]
        }
    }
}
```

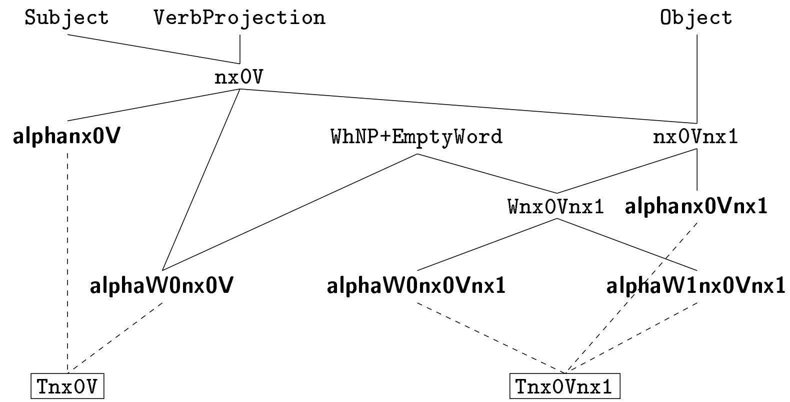

XMG - Case study - Adding fragments for extraction

1. the modified subject class is used to define the class nx0V, which is then reused in alphaW0nx0V:



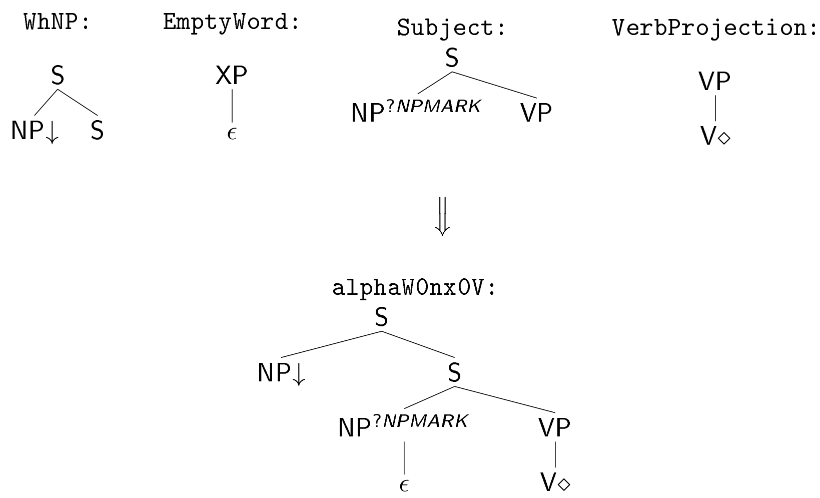
XMG - Case study - Adding fragments for extraction

1. the modified subject class is used to define the class nx0V, which is then reused in alphaW0nx0V:



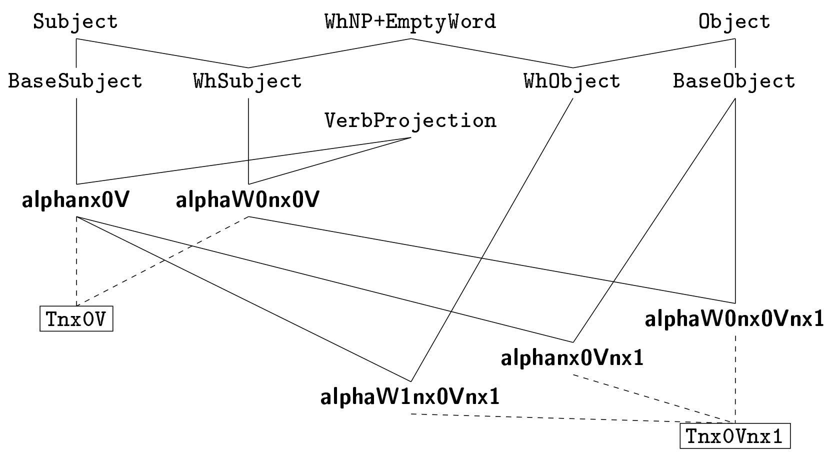
XMG - Case study - Adding fragments for extraction

2. alphaW0nx0V is directly defined through the modified subject class:



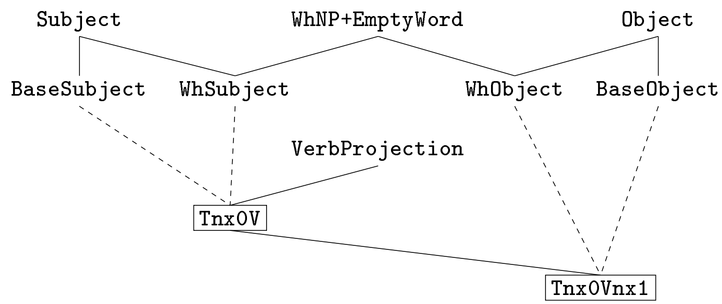
XMG - Case study - Adding fragments for extraction

2. `alphaW0nx0V` is directly defined through the modified subject class:



XMG - Case study - Adding fragments for extraction

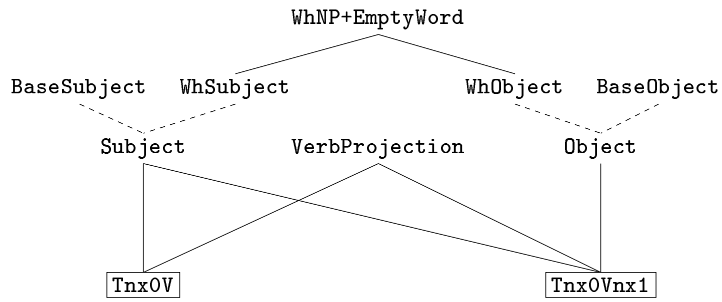
... **or** one dispenses with the evaluation of tree templates and uses only tree families:



NB: This probably makes necessary the use of **interface constraints** in order to rule out multiple Wh-extraction, i.e. to reuse the right members of tree families.

XMG - Case study - Adding fragments for extraction

... **or** one dispenses with the evaluation of tree templates and uses only **unrelated** tree families:



Along the lines of [Alahverdzhieva, 2008], no interface constraints necessary.

XMG - Summary

- XMG is a sophisticated tool for describing elementary trees and tree families in a factorized manner, i.e. based on tree fragments.
- XMG is declarative/monotonous.
- XMG is very flexible, hence it allows for very many different ways to describe the same grammar.
- How to choose among suitable metagrammars? Number of classes? Number of inheritance relations? Complexity of inheritance hierarchies?
  - ⇒ No obvious criterion for the non-trivial cases, particularly with broad coverage grammars.
  - ⇒ Considerably depends on what the grammar writer prefers ...

Alahverdzhieva, K. (2008).
XTAG using XMG. A core tree-adjoining grammar for English.
Master's thesis, University of Nancy 2 / University of Saarland.

Becker, T. (1994).
HyTAG: A New Type of Tree Adjoining Grammars for Hybrid Syntactic Representations of Free Word Order Languages.
PhD thesis, Universität des Saarlandes.

Becker, T. (2000).
Patterns in metarules for TAG.
In Abeillé, A. and Rambow, O., editors, Tree Adjoining Grammars: Formalisms, Linguistic Analyses and Processing, volume 107 of CSLI Lecture Notes, pages 331–342. CSLI Publications, Stanford.

Candito, M.-H. (1996).
A principle-based hierarchical representation of LTAGs.
In Proceedings of the 16th International Conference on Computational Linguistics (COLING 96), Copenhagen.

Crabbé, B. (2005).
Représentation informatique de grammaires d'arbres fortement lexicalisées: Le cas de la grammaire d'arbres adjoints.
PhD thesis, Université Nancy 2.

Dowty, D. R. (1979).
Word Meaning and Montague Grammar.
D. Reidel Publishing Company, Dordrecht, Boston, London.
Reprinted 1991 by Kluwer Academic Publishers.

Duchier, D., Le Roux, J., and Parmentier, Y. (2004).
The Metagrammar Compiler: An NLP Application with a Multi-paradigm Architecture.
In Second International Mozart/Oz Conference (MOZ'2004).

Gazdar, G. (1981).
Unbounded dependencies and coordinated structure.
Linguistic Inquiry, 12:155–182.

Kallmeyer, L., Lichte, T., Maier, W., Parmentier, Y., and Dellert, J. (2008).
Developing a TT-MCTAG for German with an RCG-based parser.
In (ELRA), E. L. R. A., editor, Proceedings of the Sixth International Language Resources and Evaluation (LREC'08), Marrakech, Morocco.

Parmentier, Y., Kallmeyer, L., Maier, W., Lichte, T., and Dellert, J. (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), pages 121–128, Tübingen, Germany.

Prolo, C. A. (2002).
Generating the XTAG English grammar using metarules.
In Proceedings of COLING-02, pages 814–820, Taipei. Taiwan.

Xia, F. (2001).
Automatic grammar generation from two different perspectives.
PhD thesis, University of Pennsylvania.

XTAG Research Group (2001).
A Lexicalized Tree Adjoining Grammar for English.
Technical report, Institute for Research in Cognitive Science, University of Pennsylvania, Philadelphia, PA.