Funded by Deutsche Forschungsgemeinschaft
German Research Foundation

Tree-Adjoining Grammars: Theory and implementation

Day 4: Grammar implementation with XMG

Kata Balogh & Simon Petitjean
Heinrich-Heine-Universität Düsseldorf, Carl von Ossietzky Universität Oldenburg

NASSLLI 2025

June 23 – 27, 2025

University of Washington, Seattle

Outline

Last sessions

Mon: Motivation and the basic TAG

Tue: Linguistic applications and using LTAG: syntax

Wed: Linguistic applications and using LTAG: semantics

The following sessions

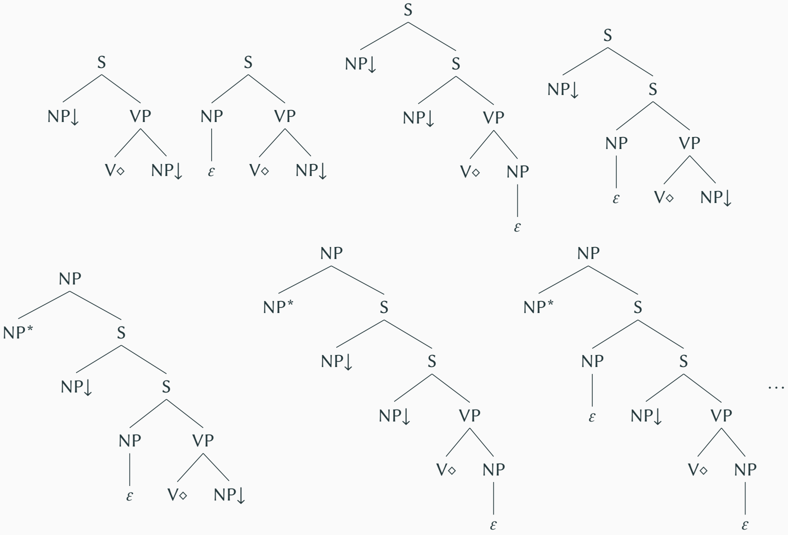
Wed: Introduction to grammar engineering and XMG

Thu: Grammar implementation with XMG

Fri: Parsing TAG

Outline

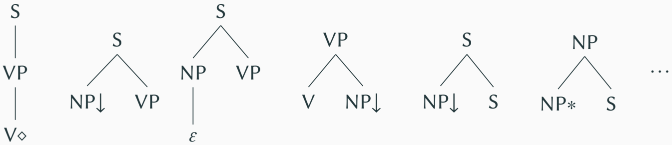
The problem: large (but highly redundant) families

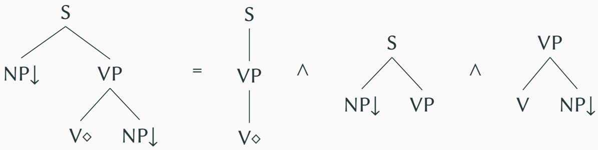


Metagrammars

- Idea: describe smaller units to capture redundancies
- Tree fragments: reusable abstractions based on linguistic (or not) generalizations
- Once the fragments are defined, the trees are created by assembling the fragments

Abstractions - Tree fragments

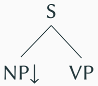




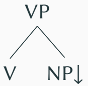
Named abstractions - Classes

`VerbalSpine` =

`CanonicalSubject` =



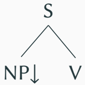
`CanonicalObject` =



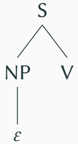
`SimpleTransitive` = `VerbalSpine` $\wedge$ `CanonicalSubject` $\wedge$ `CanonicalObject`

Expressing alternatives - Disjunction

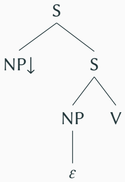
CanonicalSubject =



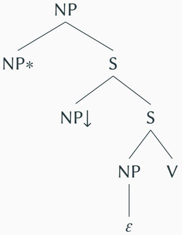
ImperativeSubject =



WhSubject =



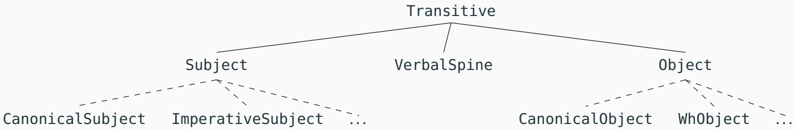
RelativeSubject =

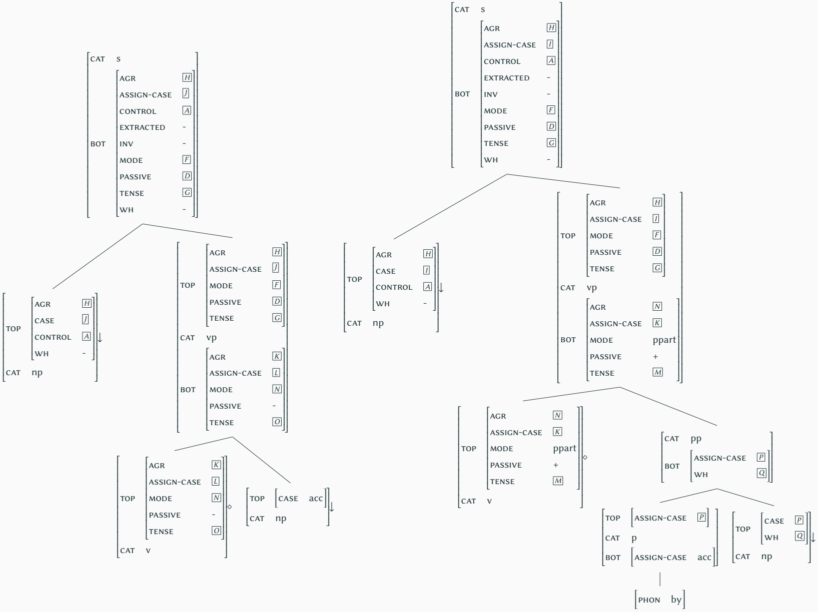


Subject = CanonicalSubject ∨ ImperativeSubject ∨ WhSubject ∨ RelativeSubject ∨ ...

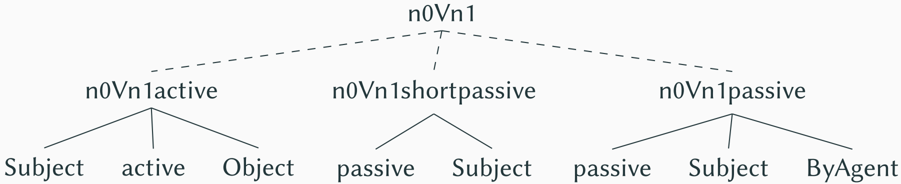
Building complex class hierarchies - Families

$$\texttt{Transitive} = \texttt{Subject} \wedge \texttt{VerbalSpine} \wedge \texttt{Object}$$





CAT s
BOT AGR H ASSIGN-CASE J CONTROL A EXTRACTED - INV - MODE F PASSIVE D TENSE G WH -
TOP AGR H CASE J CONTROL A WH - ↓
CAT np
TOP AGR H ASSIGN-CASE J MODE F PASSIVE D TENSE G
CAT vp
BOT AGR K ASSIGN-CASE L MODE N PASSIVE - TENSE O
TOP AGR K ASSIGN-CASE L MODE N PASSIVE - TENSE O ◇
CAT v
TOP CASE acc ↓
CAT np
CAT s
BOT AGR H ASSIGN-CASE I CONTROL A EXTRACTED - INV - MODE F PASSIVE D TENSE G WH -
TOP AGR H CASE I CONTROL A WH - ↓
CAT np
TOP AGR H ASSIGN-CASE I MODE F PASSIVE D TENSE G
CAT vp
BOT AGR N ASSIGN-CASE K MODE ppart PASSIVE + TENSE M
TOP AGR N ASSIGN-CASE K MODE ppart PASSIVE + TENSE M ◇
CAT v
CAT pp
BOT ASSIGN-CASE P WH Q
TOP ASSIGN-CASE P
CAT p
BOT ASSIGN-CASE acc
PHON by
TOP CASE P WH Q ↓
CAT np


n0Vn1
n0Vn1active
n0Vn1shortpassive
n0Vn1passive
Subject
active
Object
passive
Subject
passive
Subject
ByAgent

Outline

eXtensible Metagrammar (XMG): Background

- Developed at LORIA, Nancy, LIFO, Orléans and HHU, Düsseldorf.[4,7]
- Description language based on logic and constraints
- All information at `https://xmg-hhu.github.io/`

eXtensible Metagrammar (XMG): Background

- Developed at LORIA, Nancy, LIFO, Orléans and HHU, Düsseldorf.[4,7]
- Description language based on logic and constraints
- All information at `https://xmg-hhu.github.io/`

Why "eXtensible" ?

- Highly modularized[6]
- 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.[4,7]
- Description language based on logic and constraints
- All information at `https://xmg-hhu.github.io/`

Why "eXtensible"?

- Highly modularized[6]
- Dimensions with dedicated description languages and compilers (**`<syn>`**, **`<sem>`**, **`<frame>`**, **`<morph>`**, …)
- Interface using shared variables

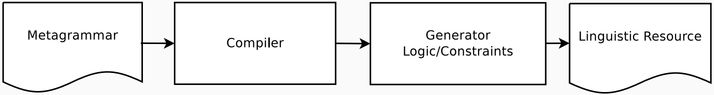
Some existing implementations using XMG:

- French: FrenchTAG[3]
- English: XTAG with XMG[1]
- German: GerTT[5]
- Arabic: ArabTAG[2]

How does it work?

XMG processing steps are as follow:

- The metagrammar is compiled: metagrammatical descriptions are 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)



Installing XMG 2

Three options, provided by the documentation:
`https://xmg-hhu.github.io/documentation`

- Follow the steps (Ubuntu), or
- Install VirtualBox and get the XMG image
- Install Docker and get the container (recommended)

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

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

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:

```
node ?S [cat=s];
node ?VP [cat=vp];
node ?NP (mark=subst) [cat=np];
?S -> ?VP;
?S -> ?NP;
?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 (... , ...+ , ,,, , ,,,+)

```
node ?S [cat=s]{
  node ?NP (mark=subst) [cat=np]
  node ?VP [cat=vp]
}
```

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

```
<syn>{
    node ?S [cat=s];
    node ?VP [cat=vp];
    node ?NP (mark=subst) [cat=n];
    ?S -> ?VP;
    ?S -> ?NP;
    ?NP >> ?VP
}
```

Syntactic nodes

Two nodes can be unified if:

- their feature structures can be unified
- their properties can be unified (except for colors, see later)

Unification of nodes happens at two different stages:

- During the execution of the code ("explicit" unification: unification operator **=** or common variable name in different classes)
- After solving: some nodes may be merged to obtain a minimal model

Minimal models

A minimal model is a model of the description where:

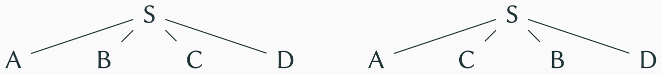
- no constraint is violated
- no additional node is created

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

```
node ?S [cat = s] ; node ?A [cat = a] ; node ?B [cat = b]
; ?S ->+ ?A ; ?S -> ?B
```

```
node ?S [cat = s] ; node ?A [cat = a] ; node ?B [cat = b]
; node ?C [cat = c] ; ?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):

```
class kicked_the_bucket
import nx0Vnx1[]
declare ?X0 ?X1
```

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

Defining abstractions

```
class Intransitive
declare ?S ?NP ?VP ?V
{
  <syn>{
      node ?S [cat=s];
      node ?VP [cat=vp];
      node ?V (mark=anchor) [cat=v];
      node ?NP (mark=subst) [cat=n];
      ?S -> ?VP; ?VP -> ?V;
      ?S -> ?NP; ?NP >> ?VP
  }
}
```

Valuation

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

```
value Intransitive
```

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, but can be accessed with the `dot` operator

Calling classes has two advantages:

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

Examples:

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

```
?C = AnotherClass[?AParameter] ; ?LocalNP = ?C.?NP
```

Classes: examples (1)

```
class a
export ?A
declare ?A ?S
{
  <syn>{
    node ?S [cat = s];
    node ?A [cat = a];
    ?S -> ?A
  }
}
```

```
class b
import a[]
declare ?B
{
  <syn>{
    node ?B [cat = b];
    ?A -> ?B
  }
}
```

Classes: examples (2)

```
class a
export ?S
declare ?A ?S
{
  <syn>{
    node ?S [cat = s];
    node ?A [cat = a];
    ?S -> ?A
  }
}
```

```
class b
import a[]
declare ?A
{
  <syn>{
    node ?A [cat = a];
    ?S -> ?A
  }
}
```

Classes: examples (3)

```
class a
export ?S
declare ?A ?S
{
  <syn>{
    node ?S [cat = s];
    node ?A [cat = a];
    ?S -> ?A
  }
}
```

```
class b
declare ?A ?Class
{
  ?Class = a[];
  <syn>{
    node ?A [cat = a];
    ?Class.?S -> ?A
  }
}
```

Classes: examples (4)

```
class a
export ?S
declare ?A ?S
{
  <syn>{
    node ?S [cat = s];
    node ?A [cat = a];
    ?S -> ?A
  }
}
```

```
class b
declare ?S ?Class
{
  ?Class = a[];
  <syn>{
    node ?S [cat = s];
    ?Class.?S -> ?S
  }
}
```

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:

```
type CAT=  {np,vp,s,n,v,det}
type MARK= {lex,anchor,subst}
type LABEL !
type PERS= [1..3]
type GEN = {m,f}
type NUM = {sg,pl}
type AGR = [gen:GEN, num:NUM]


feature cat:  CAT
feature e:    LABEL
feature pers: PERS
feature agr:  AGR

property mark: MARK
```

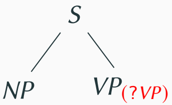
Outline

Combining tree fragments

- We know how to define tree fragments
- We have a clear idea of how they should combine
- Without additional constraints, XMG combines the fragments in all possible ways, as long as the models are minimal
- Explicitly specifying which nodes should be unified: tedious and error prone

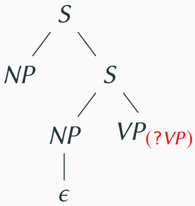
Defining a tree fragment

```
class CanonicalSubject
export VP
declare ?S ?NP ?VP
{
  <syn>{
    node ?S[cat = s];
    node ?NP[cat = np];
    node ?VP[cat = vp];
    ?S -> ?NP;
    ?S -> ?VP;
    ?NP >> ?VP }
}
```

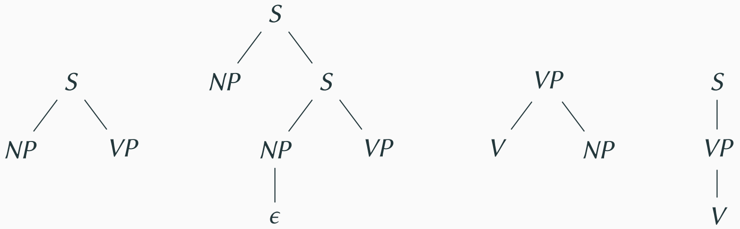


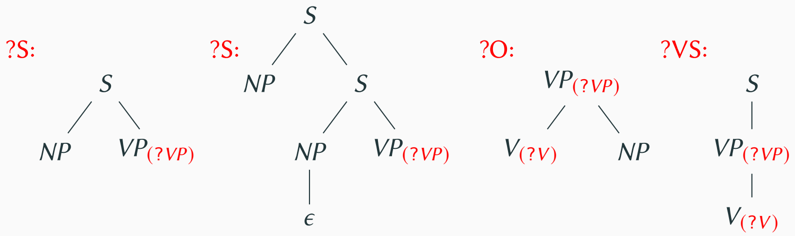
Defining a tree fragment

```
class WhSubject
export VP
declare ?S ?S1 ?NP ?NP1 ?E ?VP
{
  <syn>{
    node ?S[cat = s] ; node ?NP[cat = np] ; node ?S1 [cat = s] ;
    node ?NP1 [cat = np] ; node ?E (mark = lex) [cat = e] ;
    node ?VP [cat = vp] ;
    ?S -> ?NP ; ?S -> ?S1; ?NP >> ?S1; ?S1 -> NP1;
    ?NP1 -> ?E ; ?S1 -> ?VP; ?NP1 >> ?VP }
}
```



Assembling fragments






```
class Transitive
declare ?S ?O ?VS
{
  ?S = Subject[];
  ?O = CanonicalObject[];
  ?VS = VerbalSpine[];
  ?S.?VP = ?O.?VP;
  ?S.?VP = ?VS.?VP;
  ?VS.?V = ?O.?V
}
```

- Three last lines: not satisfying
- One solution: import the classes
- New problem: handling variable names

Handling export and disjunction

```
class Subject
export VP
declare ?CS ?WS ?VP
{
  { ?CS = CanonicalSubject[]; ?VP = ?CS.?VP }
  |
  { ?WS = WhSubject[]; ?VP = ?WS.?VP}
}
```

- Simpler and safer without the export of the **?**VP node

XMG: a local view

- Variables in XMG classes: local by default
- Advantages: avoid variable name conflicts, easier maintenance
- Disadvantages: hard to express constraints which span on several classes
- Refer to variables in foreign classes: export and import or class instantiation
- Problem: disjunction makes things more complex

Describing global constraints locally

- Aim: describe global constraints locally
- Principles: solution offered by XMG
- When do we need principles, and which ones?
- Which principles are already implemented?
- How to implement more principles?

The colors principle

- Example: previous section
- Idea: a polarity system to control fragment combinations
- A color is associated to every node
- New unification rules are given by the colors
- Proposed in **Duchier2004**

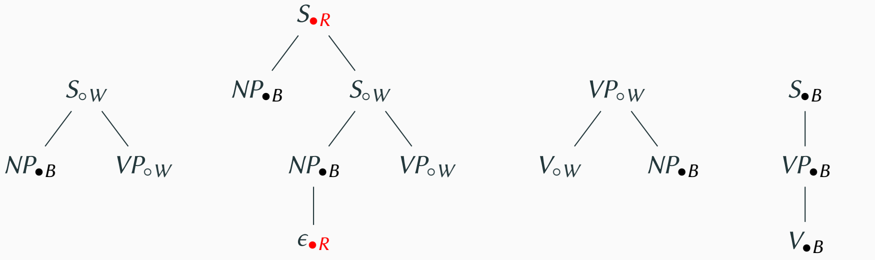
Filtering combinations with polarities

- A black node is a resource, and can be unified with white nodes
- A white node is a need, and must be unified with a black node
- A red node is saturated, and cannot be unified

	$\bullet_B$	$\bullet_R$	$\circ_W$	$\perp$
$\bullet_B$	$\perp$	$\perp$	$\bullet_B$	$\perp$
$\bullet_R$	$\perp$	$\perp$	$\perp$	$\perp$
$\circ_W$	$\bullet_B$	$\perp$	$\circ_W$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

- A valid model is composed of only red and black nodes

Colored tree fragments



CanonicalSubject *WhSubject* *CanonicalObject* *VerbalSpine*

Code for a colored tree fragment

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

class CanonicalSubject
declare ?S ?NP ?VP
{
  <syn>{
    node ?S(color=white)[cat=s];
    node ?NP(color=black)[cat=np];
    node ?VP(color=white)[cat=np];
    ?S -> ?NP;
    ?S -> ?VP;
    ?NP >> ?VP }
}
```

Assembling fragments with colors

- The `Transitive` class does not need to do explicit unifications

```
class Transitive
{
  Subject[]; CanonicalObject[]; VerbalSpine[]
}
```

Assembling fragments with colors

- The `Transitive` class does not need to do explicit unifications

```
class Transitive
{
  Subject[]; CanonicalObject[]; VerbalSpine[]
}
```

- The `Subject` class does not need to re-export variables

```
class Subject
{
  CanonicalSubject[] | WhSubject[]
}
```

Assembling fragments with colors

- The `Transitive` class does not need to do explicit unifications

```
class Transitive
{
  Subject[]; CanonicalObject[]; VerbalSpine[]
}
```

- The `Subject` class does not need to re-export variables

```
class Subject
{
  CanonicalSubject[] | WhSubject[]
}
```

- What is left?

Assembling fragments with colors

- The `Transitive` class does not need to do explicit unifications

```
class Transitive
{
  Subject[]; CanonicalObject[]; VerbalSpine[]
}
```

- The `Subject` class does not need to re-export variables

```
class Subject
{
  CanonicalSubject[] | WhSubject[]
}
```

- What is left? The class hierarchy! Only terminal classes hold descriptions

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.

[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] Ben Khelil, Chérifa, Denys Duchier, Yannick Parmentier, Chiraz Zribi & Fériel Ben Fraj. 2016. **ArabTAG: from a handcrafted to a semi-automatically generated TAG.** In *Proceedings of the 12th international workshop on tree adjoining grammars and related formalisms (TAG+12)*, 18–26. Düsseldorf, Germany. https://aclanthology.org/W16-3302.

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

[4] 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/.

[5] 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.

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

[7] Petitjean, Simon, Denys Duchier & Yannick Parmentier. 2016. **XMG 2: Describing Description Languages.** In *Logical aspects of computational linguistics. celebrating 20 years of lacl (1996–2016) 9th international conference, lacl 2016, nancy, france, december 5-7, 2016, proceedings 9*, 255–272.