Tree Adjoining Grammars: XMG

Laura Kallmeyer & Simon Petitjean

HHU Düsseldorf

WS 2015
13.11.2015
Introduction

- Large scale Tree Adjoining Grammars are composed of thousands of trees
- Manual description by an expert: time consuming
- Automatic methods: need a corpus
- Precise resources, with easier development and maintenance: semi automatic methods
Our task

- Implement a TAG and use it to parse sentences
- Structure of the grammar: similar to XTAG (3 levels)
- Generate the trees: XMG
- Morphological/syntactic lexicons: lexConverter
- Parser: TuLiPA
Semi automatic methods: Metarules

- XTAG: Metarules to generate tree families from “standard” tree
- A metarule modifies a tree (adds/removes nodes) to create a new one
- Process: apply all possible metarules until a fixpoint is reached
Metarules for LTAG: Example

\textbf{Tnx0nx1:}

\[ \alpha W_0 nx_0 V nx_1 \]
\[ \alpha nx_1 V b y nx_0 \]

\textbf{extraction}

\textbf{active-passive alternation}

\[ \alpha W_1 nx_1 V b y nx_0 \]
Semi automatic methods: Metagrammars

- MetaGrammatical Approach ([Candito, 1996]): linguistic description of the grammar
  - Instead of writing rules, define parts of rules
  - The rule fragments come in a 3-dimension hierarchy (subcategorization, syntactic functions redistribution, final functions realizations)
  - All the fragments from dimension 1 are combined with the ones from dimension 2
  - The resulting fragments are combined with the ones in dimension 3
Languages can be described at several levels: syntax, morphology, semantic, prosody, discourse, . . .

Different formalisms have been proposed: Head-driven phrase structure grammar, Lexical Functional Grammar, Tree Adjoining Grammar, Categorial Grammar, Paradigm Function Morphology, Network Morphology, . . .

eXtensible MetaGrammar (XMG): MetaGrammar compiler initially used to create large scale Tree Adjoining Grammars ([Joshi et al., 1975]) and Interaction Grammars ([Perrier, 2000])
Description tools

- Need to use a description language, adapted to the task

- Choosing a tool (LKB, XLE, ...): definitive, no way to adapt the description language

- Fit language to linguistic intuition

Slogan (XMG-2)

The user should not have to adapt to the tool
The tool should adapt to the user
XMG ([Crabbé et al., 2013])

Ambition

- Arbitrarily many levels of linguistic description (syntax, semantics, ...): dimensions
- Affect a Domain Specific Language (DSL) to each one of these levels

Methodology

- Declarative definition of rule fragments (classes)
- Combination of rule fragments with logical operators
Transitive verbs (in French)

Sally chante une chanson

la chanson que Sally chante

Elle chante une chanson

Sally qui chante une chanson
Transitive verbs (in French)

- chante la
- c’est la chanson que Sally chante
- chante une chanson
- Sally la chante
XMG: example

Laura Kallmeyer, Simon Petitjean (Uni)
XMG: example

```
CanSubj

Active

RelObj

CanObj

CanSubj ∧ Active ∧ (RelObj ∨ CanObj)
```
XMG: a MetaGrammar “compiler”

XMG is usually called a MetaGrammar compiler

- Compiler: Source code $\rightarrow$ Executable code
- XMG: Metagrammar (Grammar description) $\rightarrow$ Grammar

To make it short:

- A grammar is a description of a language
- A metagrammar is a description of a grammar
Generation steps

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

Contribution

User

Metagrammar

Meta Compiler

Specification (YAML)

Bricks Library

Programmer

Meta Executor

Executor

Generator Logic/Constraints

Specific VM

Base VM

Linguistic Resource

Meta Compilation

Compilation

Generation

Laura Kallmeyer, Simon Petitjean (Uni)
XMG-2: Architecture (relevant part for this class)
Plan

1. Introduction
2. Getting started
3. The control language
4. Describing trees
5. Minimal models
6. Defining abstractions
7. Conclusion
Installing XMG

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

- Follow the steps (Ubuntu), or
- Install VirtualBox and get the XMG image
Installing contributions

- XMG bricks are distributed as contributions

- Making a contribution available is done with the install command

```bash
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
```
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

\[
\begin{align*}
\text{Class} & \; := \; \text{Name} \rightarrow \text{Content} \\
\text{Content} & \; := \; \langle \text{Dimension}\rangle\{\text{Description}\} \mid \text{Name} \mid \text{Content} \lor \text{Content} \mid \text{Content} \land \text{Content}
\end{align*}
\]
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 (\(\rightarrow\), \(\rightarrow^+\), \(\rightarrow^*\)) and precedence operators (\(\gg\), \(\gg^+\), \(\gg^*\))

- Combining these statements: with logical operators (\(;\) and \(\mid\))

Example:

```plaintext
node ?S [cat=s];
node ?VP [cat=vp];
node ?V (mark=anchor) [cat=v];
node ?NP (mark=subst) [cat=n];
?S \rightarrow ?VP;
?VP \rightarrow ?V;
?S \rightarrow ?NP;
?NP \gg ?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 (\( \ldots, \ldots+\), \(\ldots, \ldots, \ldots\), \(\ldots, \ldots\))

```plaintext
node ?S [cat=s] {
    node ?NP (mark=subst) [cat=np]
    node ?VP [cat=vp] {
        node ?V (mark=anchor) [cat=v]
    }
}
```
Describing trees

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 ?V (mark=anchor) [cat=v];
  node ?NP (mark=subst) [cat=n];
  ?S -> ?VP;
  ?VP -> ?V;
  ?S -> ?NP;
  ?NP >> ?VP
}
```
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
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?


Which set of constraints leads to the following minimal models?

```
S  
|   |
| A | B |
|   |
| C | D |
```

```
S  
|   |
| A | C |
|   |
| B | D |
```
Defining abstractions

Classes allow to:

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

Examples (just headers):

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

class nx0Vnx1
```
Defining abstractions

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

}
}
```

Valuation

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

```plaintext
value Intransitive
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:

- `CanObj[] | RelObj[]`
- `?C=Class[?X]`
```java
class a
export ?A
declare ?A ?S
{
  <syn>{
    ?S -> ?A
  }
}
}

class b
import a[]
declare ?B
{
  <syn>{
    ?B -> ?A
  }
}
```
class a
export ?S
declare ?A ?S
{
    <syn>{
        ?S -> ?A
    }
}
}

class b
import a[]
declare ?A
{
    <syn>{
        ?S -> ?A
    }
}

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 $T=[a_1:t_1, \ldots , a_n:t_n]$
- Interval type: type $T=[1..3]$
- Unspecified type: type $T!$
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
```
Conclusion

**XMG-NG**

- Different levels of linguistic description (syntax, semantics, ...): dimensions
- A Domain Specific Language (DSL) for each one of these levels
- A specific solver for the descriptions in each dimension
- The `<syn>` dimension: tree descriptions and tree solver (minimal models)

**Methodology**

- Write a metagrammar: create abstractions on rules and combine these abstractions with logical operators
- Compile the metagrammar to generate the grammar

**For any question**

- dokufarm.phil.hhu.de/xmg

