Metagrammars as Logic Programs

Denys Duchier, Yannick Parmentier, Simon Petitjean

Metagrammars As Logic Programs

Denys Duchier, Yannick Parmentier, and Simon Petitjean

LIFO, Université d'Orléans, Bâtiment 3IA
6, Rue Léonard De Vinci - BP 6759
F-45067 Orléans Cedex 2, France,

firstname.lastname@univ-orleans.fr

Abstract. In this paper, we introduce the eXtensible MetaGrammar (XMG), which corresponds to both a language for specifying formal grammars, and a compiler for this language. XMG has been developed over the last decade to provide linguists with a declarative and yet expressive way to specify grammars. It has been applied to the design of actual tree-based grammars for French, German or English. XMG relies on a modular architecture, which makes it possible to extend the formalism with additional levels of descriptions and / or linguistic properties. Thus, on top of syntax, XMG can also be used for the description of other linguistic information such as semantics, or morphology (the latter being currently explored for Ikota, an African language spoken in Gabon).

1 Introduction

Since Chomsky's seminal work on generative grammar [1], many formal systems have been proposed to describe the syntax of natural language (see *e.g.* [2]). These mainly differ in terms of expressivity and computational complexity, and generally rely either on rewriting rules (*e.g.* Tree-Adjoining Grammar), or on constraints (*e.g.* Head-driven Phrase Structure Grammar).[1]

An interesting family of formal grammars are *lexicalized* grammars [3]. Such grammars associate each elementary structure (*i.e.* grammar rule) with a lexical item (called anchor). Lexicalized grammars offer two main advantages: firstly, the grammar can be seen as a function mapping lexical items (*i.e.* words) with *uninstantiated* grammatical structures (the grammar is then called lexicon). Secondly, a subgrammar can be extracted from the input grammar according to the sentence to parse, thus speeding up parsing.

An example of lexicalized grammar is Lexicalized Tree-Adjoining Grammar (LTAG). In this formalism, the grammar is made of (thousands of) uninstantiated elementary trees (called tree *templates*), where the leaf nodes contain at least one anchor node (labelled with $\diamond$). These anchor nodes are attached to adequate lexical items at parsing. As an illustration, consider Fig. 1 depicting two tree templates to be anchored with a transitive verb such as *manger* (to eat).

[1] We do not discuss the distinction between constituency and dependency grammar here, nonetheless the latter can be seen as a constraint-based specification of syntax.

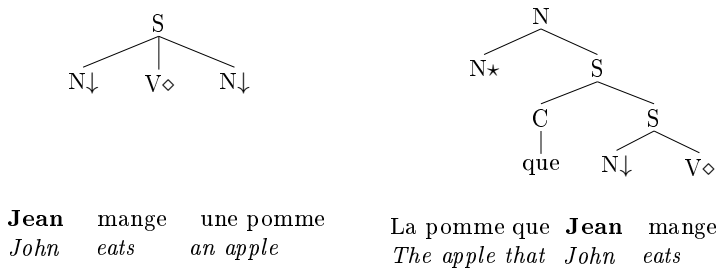

Fig. 1. Elementary structures of an LTAG

From a linguistic point of view, lexicalized grammars allow to express generalizations over lexical entries by gathering tree templates, whose anchor have similar syntactic properties, into *tree families*. From a computational point of view, lexicalized grammars are made of a huge number of structures, due to redundancy within the lexicon (*e.g.* tree templates sharing common subtrees).

The concept of *metagrammar* was introduced by *Candito* [4] in order to deal with structural redundancy by capturing generalizations over tree templates. Instead of directly describing the syntax of language via a formal grammar, the linguist *specifies* the structures of this formal grammar using a dedicated framework. This specification of the grammar is called a metagrammar and is automatically processed to generate the grammar. Many metagrammatical frameworks have been proposed for LTAG [4,5,6,7]. Here we introduce one of these, namely eXtensible MetaGrammar (XMG) [6]. XMG differs from other metagrammar approaches by its declarative specification language, and its modular architecture. The latter made it possible to extend the concept of metagrammars to other levels of description (*e.g.* morphology) and linguistic principles (*e.g.* constraints on word order), as we shall see below.

2 The XMG language

As mentioned above, the XMG language allows for a declarative specification of linguistic structures (including tree descriptions). More precisely, XMG offers a unification-based language *à la* Prolog to specify what a grammar is. This specification is then processed by the XMG compiler in order to produce a computational grammar (*e.g.* an LTAG), which can be saved in an XML file.

Capturing redundancy using abstractions. XMG relies on the concept of *abstraction* to allow the linguist to refer to reusable grammatical units (*e.g.* (combinations of) tree fragments for LTAG). Formally, an XMG specification corresponds to declarative *rules*, which can be defined using the following abstract syntax:

$$\begin{aligned} \mathit{Rule} &:= \mathit{Name} \rightarrow \mathit{Content} \\ \mathit{Content} &:= \mathit{Contribution} \mid \mathit{Name} \mid \mathit{Content} \vee \mathit{Content} \mid \mathit{Content} \wedge \mathit{Content} \end{aligned}$$

Here, *Contribution* refers to a linguistic fragment of information of a given type (*e.g.* syntax), to be accumulated either conjunctively or disjunctively. Such a

fragment is specified using a dedicated description language (*e.g.* a tree description language when describing syntax with LTAG). This language relies on unification variables to share information between distinct XMG rules (*i.e.* distinct grammatical units) or between distinct contributions (*i.e.* between syntax and semantics). The scope of these variables is by default restricted to the rule, but can be extended via *import / export* declarations. As a toy example of these variables and of XMG concrete syntax, consider the rules `CanonicalSubject` and `Subject` below, the latter specifies a generalization over the two possible realizations of a subject shown in Fig. 1 (-> is dominance and >> precedence).

```
class CanonicalSubject          %% (comment) a class is an XMG rule in the abstract syntax
export ?x ?y
declare ?x ?y ?z ?u
{<syn>                          %% contribution of type <syn>
  { node ?x [cat=S] ; node ?z [cat=N] ; node ?y (type=anchor)[cat=V] ; node ?u [cat=N] ;
    ?x -> ?z ; ?x -> ?y ; ?x -> ?u ; ?z >> ?y ; ?y >> ?u }
}

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

Towards user-defined description languages. Metagrammars bring interesting insights in grammar engineering by offering an abstract view on language, made of combinations of grammatical units. So far, these units were described using a set of hard-coded descriptions languages. To reach extensibility, we are exploring another approach: permitting user-defined description languages (similarly to the grammar, these must be described). Some parts of the compiler thus have to be generated automatically.

3 The XMG compiler

General architecture. As mentioned above, the XMG language is nothing else than a logic language. Its compiler thus share some features with a compiler for logic programs. First, the classes composing the metagrammar (defined using the XMG language introduced above) are converted into clauses of an Extended Definite Clause Grammar (EDCG) [8], which corresponds to a DCG having multiple accumulators. This underlying EDCG explicits the accumulation of contributions of multiple types (*e.g.* syntax, semantics). Then, this EDCG is evaluated according to axioms defined in the metagrammar (comparable to Prolog queries). This produces a list of tuples of contributions (the arity of these tuples is the number of contribution types). Finally, each tuple of this list is optionally post-processed. For instance, tuples whose syntactic contribution is a tree description are fed to a solver in order to produce syntactic trees. During this solving step, it is possible to apply linguistic well-formedness principles (these can use information from other contributions of the tuple).

XMG 2. The first version of XMG (XMG 1.x) was developed between 2003 and 2010 in the Oz programming language, and included only three description languages: one for specifying syntactic trees (either LTAG tree templates or Interaction Grammar tree descriptions), one for specifying semantic representations, and one for specifying the syntax / semantics interface. The development

of a new version of XMG from scratch in YAP Prolog started in 2010, in order to extend XMG with the ability to define an arbitrary number of types of contributions (and thus of user-defined description languages).[2]

4 Current state and future work

XMG can be used to describe tree structures, feature structures, predicates, or *properties* of the Property Grammar formalism. Version 2 of the XMG language superseeds Version 1 (being backward-compatible). XMG 2 can be used to compile grammars designed with XMG 1, including the French LTAG and French Interaction Grammar, whose XMG metagrammars are available on-line (along with toy examples of XMG input / output).[3] When describing LTAG tree templates, XMG 2 offers specific linguistic principles, namely ordering between sister nodes, uniqueness of a given node label, and node merging via polarities.

XMG 2 is being actively developed in order to allow for cross-framework grammar engineering, in the lines of [9], but also for linguistic experimentation by defining dynamically its own grammar formalism as mentioned in Section 2.

XMG 2 has been used recently to describe the morphology of verbs in Ikota, an agglutinative Bantu language spoken in Gabon [10]. The idea behind this work is to specify morphemes as contributions in terms of lexical phonology and inflection (morpho-syntactic features). In a next step, we plan to extend this metagrammar (*i.e.* this abstract linguistic account of morphology) to syntax.

References

1. Chomsky, N.: Syntactic Structures. Mouton, The Hague (1957)
2. Abeillé, A.: Les Nouvelles Syntaxes. Armand Colin, Paris (1993)
3. Schabes, Y., Abeillé, A., Joshi, A.K.: Parsing strategies with 'lexicalized' grammars: application to Tree Adjoining Grammars. In: 12th COLING. (1988) 578–583
4. Candito, M.: A Principle-Based Hierarchical Representation of LTAGs. In: 16th COLING. (1996) 194–199
5. Xia, F.: Automatic Grammar Generation from two Different Perspectives. PhD thesis, University of Pennsylvania (2001)
6. Duchier, D., Le Roux, J., Parmentier, Y.: The Metagrammar Compiler: An NLP Application with a Multi-paradigm Architecture. In: MOZ. (2004) 175–187
7. Villemonte De La Clergerie, É.: Building factorized TAGs with meta-grammars. In: TAG+10, New Haven, CO, United States (2010) 111–118
8. Van Roy, P.: Extended dcg notation: A tool for applicative programming in prolog. Technical report, Technical Report UCB/CSD 90/583, UC Berkeley (1990)
9. Duchier, D., Parmentier, Y., Petitjean, S.: Cross-framework Grammar Engineering using Constraint-driven Metagrammars. In: CSLP. (2011) 32–43
10. Duchier, D., Magnana Ekoukou, B., Parmentier, Y., Petitjean, S., Schang, E.: Describing Morphologically-rich Languages using Metagrammars: a Look at Verbs in Ikota. In: 4th Workshop on African Language Technology - LREC. (2012)

[2] Both implementations (XMG 1.x and XMG 2.x) are freely available on-line at `https://sourcesup.renater.fr/xmg` and `https://launchpad.net/xmg` respectively.

[3] `https://sourcesup.cru.fr/scm/viewvc.php/trunk/METAGRAMMARS/?root=xmg`