The Metagrammar Compiler: An NLP Application with a Multi-paradigm Architecture

Denys Duchier, Joseph Le Roux, and Yannick Parmentier

LORIA
Campus Scientifique,
BP 239,
F-54 506 Vandœuvre-lès-Nancy, France
{duchier,leroux,parmenti}@loria.fr

Summary. The concept of metagrammar has been introduced to factorize information contained in a grammar. A metagrammar compiler can then be used to compute an actual grammar from a metagrammar. In this paper, we present a new metagrammar compiler based on 2 important concepts from logic programming, namely (1) the Warren's Abstract Machine and (2) constraints on finite set.

1 Introduction

In order to develop realistic NLP applications and support advanced research in computational linguistics, large scale grammars are needed. By the end of 90's, several such grammars had been developed by hand; especially for English [1] and French [2].

Unsurprisingly, wide-coverage grammars become increasingly hard to extend and maintain as they grow in size and scope. There is often grammatical information which cannot be adequately modularized and factorized using the facilities offered by standard grammar formalisms. As a consequence, grammar rules become distressingly rife with structural redundancy and any modification frequently needs to be repeated in many places; what should be a simple maintenance intervention turns into a chore which is both work intensive and error prone.

For these reasons, and others, a new methodology for grammar development has emerged that is based on the compilation of meta-descriptions. These meta-descriptions should help express simply linguistically relevant intuitions, as well as mitigate the redundancy issue through better means of factorizing the information present in the rules of the grammar.

In this paper, we present a system designed for generating a wide-coverage Tree Adjoining Grammar (TAG) from such a meta-description (generally called a metagrammar). Our proposal is especially novel in that it adopts a resolutely multi-paradigmatic approach: it combines (1) an object-oriented specification language for abstracting, structuring, and encapsulating fragments of grammatical information, (2) a logic programming backbone for expressing the combinations and non-deterministic choices of the metagrammatical specification, (3) a constraint-based back-end to resolve underspecified combinations.

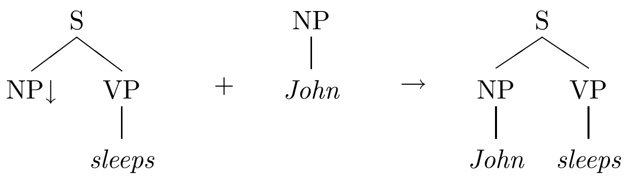
2 Tree Adjoining Grammars

In this section, we informally introduce the notion of a *tree adjoining grammar* (TAG).

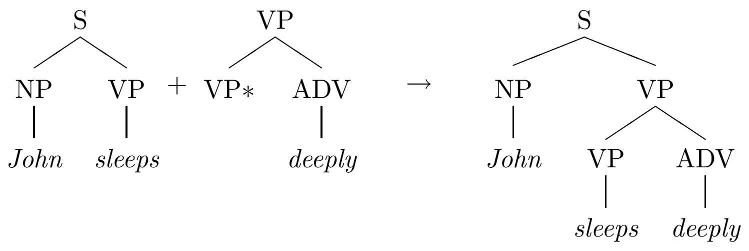
A grammar is a formal device used to describe the syntax of natural (or artificial) languages. While details vary, at heart, a grammar consists in the stipulation of a finite number of building blocks and a finite number of operations to combine them together.

In the Chomskian tradition of context free grammars, the building blocks are *production rules* and the only operation is the expansion of a non-terminal by application of a matching production rule.

In TAG, the building blocks are tree fragments, and there are two operations to combine them called *substitution* and *adjunction*. Substitution plugs one tree fragment into a matching leaf, marked for substitution (i.e. marked with ↓) of another tree fragment:



Adjunction splices in one tree fragment, from root to foot node (the latter marked with ∗), in place of a matching node in another tree fragment:



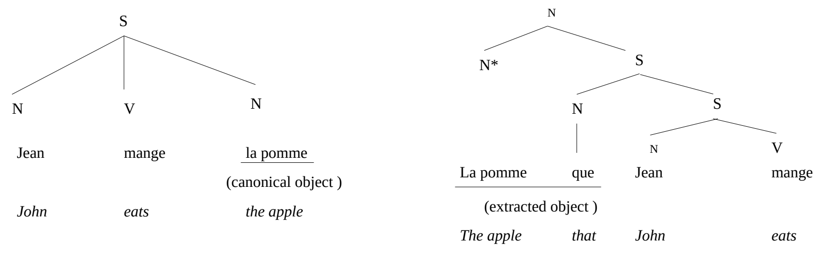
TAGs are used as a formalism designed for describing natural language syntax because of their linguistic properties [3]. A precise introduction to TAG is given in [4]. TAGs belong to the family of so-called *mildly context-sensitive* grammars as their generative capacity is larger than just the context free languages.

3 The concept of Metagrammar

A TAG consists of a very large number (thousands) of tree fragment schemata. The reason for this large number of trees is that basically a TAG enumerates for each word all its possible *patterns* of use. Thus, not only can a verb be used in many ways (e.g. active vs. passive), but its arguments can also be realized in various ways such as direct object vs. clitic vs. extracted as illustrated in[1]:

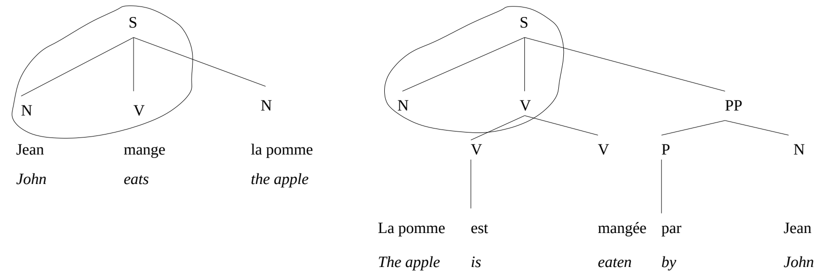
- Jean mange *la pomme*
- Jean *la* mange
- la pomme *que* Jean mange

Thus, while a TAG will contain verbal tree fragments for these two constructions:



they actually both derive from the same linguistic intuitions about the possibilities for realizing a verb and its arguments. A formalism which only allows us to write tree fragments is insufficient to also express this higher-level view of how tree fragments actually arise simply from linguistic regularities governing how verbs and their arguments can be realized.

Adopting a more engineering-minded view, we arrive at a dual perspective on essentially the same issue: current large-scale TAG suffer from a high degree of structural redundancy as illustrated in:



[1] In this paper, the tree schematas are inspired by [2] and are characterised by the absence of VP or NP nodes.

In order to ease development and maintenance, it would again be advantageous to be able to factorize such common chunks of grammatical information. Thus we have illustrated two important motivations for the factorization of grammatical information: (1) *structure sharing* to avoid redundancy [5], and (2) *alternative choices* to express diathesis such as active, passive. Attempts to address these issues lead to the notion of *metagrammar*, i.e. to formalisms which are able to *describe* grammars at a higher-level of abstraction and in more modular ways.

4 Existing Metagrammars and Compilers

The notion of metagrammar as a practical device of linguistic description (as opposed to merely increasingly expressive grammar formalisms) has a fairly short history, but is now rapidly gaining support in the linguistic community. In this section, we first review the seminal work of Candito [6], then the revised improvements of Gaiffe [7], finally leading to our own proposal[2].

4.1 A framework based on 3 linguistic dimensions

The first implementation of a metagrammar (MG) compiler was realized by Marie-Hélène Candito [6]. It laid down the bases of the MG concept which are:

- a MG is a modular and hierarchical representation of the trees of a TAG
- the hierarchy is based on linguistic principles

This compiler was used at the Université Paris 7 to automate the writing of the French TAG, was coded in Common LISP, and dealt with verbal trees.

Candito's MG methodology stipulates three dimensions, each containing hierarchically organized classes:

1. the first dimension provides the *initial subcategorization frame* (e.g. active transitive verb) which reflects the number of arguments of a verb and their positions.
2. the second dimension handles the *redistribution of syntactic functions*, i.e. the modifications of the function of the arguments defined in the 1st dimension (e.g. active becoming passive).
3. the third dimension expresses the different *realizations* for each syntactic function (canonical, cleft, etc).

Classes typically contain some topological information (e.g. tree descriptions [9]). The combination operation picks one class from the 1st dimension, one class from the 2nd dimension and n classes from the 3rd dimension, where n is the number of realized arguments of the verb. Figure 1 illustrates the idea of this 3-dimensional hierarchy and offers an example of a generated TAG tree. Candito's approach has the following drawbacks:

[2] One should also consult Xia's work [8] to have a more complete view of the process of automatic generation of TAGs.

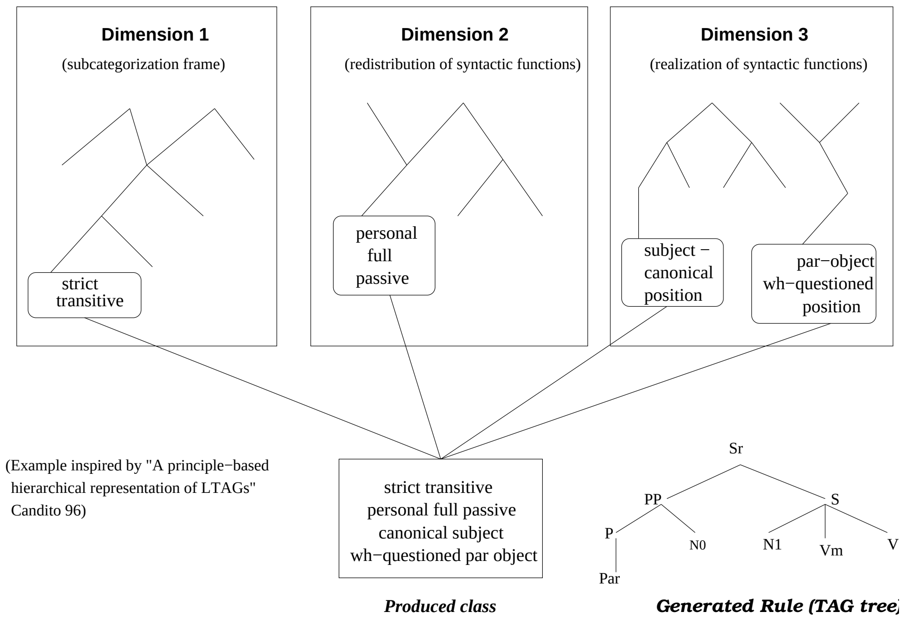

Dimension 1
(subcategorization frame)
strict transitive
Dimension 2
(redistribution of syntactic functions)
personal full passive
Dimension 3
(realization of syntactic functions)
subject − canonical position
par−object wh−questioned position
(Example inspired by "A principle−based hierarchical representation of LTAGs" Candito 96)
strict transitive
personal full passive
canonical subject
wh−questioned par object
Produced class
Sr
PP
S
P
N0
N1
Vm
V
Par
Generated Rule (TAG tree

Fig. 1. 3-dimensional hierarchy.

1. class evaluation is non monotonic, as some information can be erased during the compilation process, *e.g.* in agentless passive.
2. there is no clean separation between the knowledge encoded in the meta-description and the procedural knowledge encoded in the compiler. As a result (a) the compiler is hard to extend, and (b) you cannot define meta-descriptions with more than 3 dimensions.
3. the combination mechanism wildly attempts all possible class crossings. It is difficult to achieve enough control to avoid undesirable combinations.

4.2 A framework based on the concept of Needs and Resources

To address the issues identified with Candito's approach, Bertrand Gaiffe *et al.* at LORIA developed a new MG compiler [7], in Java, with the following properties:

- the process of class evaluation is monotonic;
- you may define an arbitrary number of dimensions instead of being limited to the strictly 3-dimensional approach of Candito.

In this implementation, a MG corresponds to several hierarchies of classes in multiple inheritance relation. The classes contain partial tree descriptions and/or node equations. The novelty is that classes can be annotated with *Needs* and *Resources*. For instance, the class for a transitive verb bears the annotation that it needs a subject and an object, while a class for a nominal construction would indicate that it supplies e.g. a subject. The combination process that produces the TAG grammar is entirely driven by the idea of matching needs and resources. However, there are still some drawbacks:

1. while the notion of needs and resources generalizes Candito's approach and allows to drive the combination process more accurately, it still exhibits the same practical drawback, namely that too many useless crossings must be explored. This problem, also present in Candito, comes from the lack of separation between the realization of *structure sharing* and the expression of *alternative choices*.
2. All node names have global scope (same as with Candito). In wide-coverage grammars, name management, and the discovery and handling of name conflicts become unrealistically difficult.
3. Since names have global scope, it is not possible to instantiate the same class more than once in a complex crossing because the names of the two instances would clash. This poses problems e.g. for constructions requiring two prepositional arguments.

4.3 A framework based on nondeterminism and underspecification

Our approach realizes a methodology developed jointly with Benoit Crabbé at LORIA and aimed at large TAG lexica [10]. Crabbé's essential insight is that instead of matching nodes very strictly by names, we can use some form of underspecification. The main requirements are:

1. There are no transformations, such as deletion, to compute a special form (*i.e.* passive, middle, extracted... for verbs) from the canonical form (basic active frame). Only alternative constructions are given. This is an important point (see [11]) since it makes our formalism monotonic and declarative.
2. Instead of being given names, nodes are assigned colors, which basically correspond again to a notion of needs and resources, that constrain how they can (or must) be matched.
3. Linguistically motivated global well-formedness principles can be stated that limit the admissibility of resulting combinations.

We depart from previous approaches on the following points:

1. The MG uses a logical language of conjunctions and disjunctions to express directly how abstractions are to be combined. With Candito, the mechanism is completely external to the MG. With Gaiffe, it is still implicit, but driven by needs and resources. Our method brings us consequent time savings during grammar generation.
2. The MG consists of classes arranged in a multiple inheritance hierarchy. Each class can introduce local identifiers, and their scope in the hierarchy can be managed with precision using import and export declarations. Renaming is supported.

3. We expressedly wanted our MG to handle not only syntax, but also semantics. For this reason, our design is multi-dimensional, where each dimension is dedicated to a descriptive level of linguistic information. To our knowledge, ours is the first MG compiler to offer such a naturally integrated syntax/semantics interface.
4. Our design is not TAG-specific and can be instantiated differently to accommodate other formalisms. It is currently being adapted for Interaction Grammars [12].

Our tool is implemented in Mozart/Oz and has been used by linguists at LORIA to develop French wide-coverage grammars.

5 A new Metagrammatical Formalism

In this section, we first introduce the logical core of our formalism using the paradigm of *Extended Definite Clause Grammars* [13]. Then we introduce the object-oriented concrete level and show how it can be translated into this core.

5.1 Logical core

Metagrammar as grammar of the lexicon.

There is a well-known descriptive device which offers abstractions, alternations, and compositions, namely the traditional generative grammar expressed as production rules. In our MG application, the elements which we wish to combine are not words but e.g. tree descriptions, yet the idea is otherwise unchanged:

$$\begin{aligned} \mathit{Clause} \quad &::= \quad \mathit{Name} \rightarrow \mathit{Goal} && (1)\\ \mathit{Goal} \quad &::= \quad \mathit{Description} \;\mid\; \mathit{Name} \;\mid\; \mathit{Goal} \vee \mathit{Goal} \;\mid\; \mathit{Goal} \wedge \mathit{Goal} && (2)\end{aligned}$$

We thus start with a logical language which can be understood as a *definite clause grammar* (DCG) where the terminals are tree *Descriptions*. We can already write abstractions such as:

$$\begin{aligned} \mathit{TransitiveVerb} \quad &\rightarrow \quad \mathit{Subject} \wedge \mathit{ActiveVerb} \wedge \mathit{Object}\\ \mathit{Subject} \quad &\rightarrow \quad \mathit{CanonicalSubject} \vee \mathit{WhSubject}\end{aligned}$$

Tree description language.

We adopt a tree *Description* language that is based on dominance constraints:

$$\mathit{Description} \; ::= \; x \rightarrow y \;\mid\; x \rightarrow^{*} y \;\mid\; x \prec y \;\mid\; x \prec^{+} y \;\mid\; x[f{:}E] \;\mid\; x(p{:}E) \qquad (3)$$

x, y range over node variables, $\rightarrow$ represents immediate dominance, $\rightarrow^{*}$ its reflexive transitive closure, $\prec$ is immediate precedence, and $\prec^{+}$ its transitive closure. $x[f{:}E]$ constrains feature f on node x, while $x(p{:}E)$ specifies its property p, such as `color`.

Accumulations in several dimension.

When the meta-grammar terminals are syntactic tree fragments, we have a meta-grammar that can describe syntax, but we also want to support other descriptive levels such as semantics. Basically, we want to accumulate descriptive fragments on multiple levels.

This can be done simply by reaching for the formalism of *extended definite clause grammars* (EDCG) [13]: where a DCG has a single implicit accumulator, an EDCG can have multiple named accumulators, and the operation of accumulation can be defined arbitrarily for each one. In (2), we replace *Description* with:

$$\mathit{Dimension} \mathrel{+}= \mathit{Description}$$

which explicitly accumulates *Description* on level *Dimension*. In our application to TAG we currently use 3 accumulators: **syn** for syntax, **sem** for semantics, and **dyn** for an open feature structure accumulating primarily morpho-syntactic restrictions and other items of lexical information.

Managing the scope of identifiers.

One of our goals is to support a concrete language with flexible scope management for identifiers. This can be achieved using explicit imports and exports. We can accommodate the notion of exports by extending the syntax of clauses:

$$\mathit{Clause} \quad ::= \quad \langle f_1{:}E_1, \ldots, f_n{:}E_n \rangle \Leftarrow \mathit{Name} \quad \rightarrow \quad \mathit{Goal} \tag{4}$$

where $\langle f_1{:}E_1, \ldots, f_n{:}E_n \rangle$ represents a record of exports. Correspondingly, we extend the abstract syntax of a *Goal* to replace the invocation of an abstraction *Name* with one that will accommodate the notion of imports:

$$\mathit{Var} \Leftarrow \mathit{Name} \tag{5}$$

To go with this extension, we assume that our expression language permits feature lookup using the dot operator, so that we can write $\mathit{Var}.f_k$, and that *Goal*s can also be of the form $E_1 = E_2$ to permit equality constraints. Finally, we allow writing *Name* instead of $\_ \Leftarrow \mathit{Name}$ when the exports are not of interest.

5.2 Object-oriented concrete syntax

A MG specification consists of (1) definitions of types, features and properties, (2) class definitions, (3) valuations. For lack of space, we omit concrete support for defining types, typed features attaching morpho-syntactic information with nodes, and properties annotating nodes with e.g. color or an indication of their nature (anchor, substitution node, foot-node...). We introduce the concrete syntax for class definitions by example, together with its translation into the logical core.

Class definitions.

Classes may actually take parameters, but we omit this detail here. A class may introduce local identifiers, and export some of them, and has a body which is just a *Goal*. Here is an example on the left, and its translation into the logical core on the right:

```
class A
  define ?X ?Y
  export X
{ X=f(Y) }
```

$$\equiv \quad \langle \texttt{X}{:}X\rangle \Leftarrow A \quad \rightarrow \quad X = f(Y)$$

Inheritance is expressed with import declarations. Importing class A in the definition of class B is very much like instantiating (calling) it in B's body, except for scope management: when A is imported, all its identifiers are made available in B's scope and automatically added to B's exports.

$$\begin{array}{rcccc} \texttt{class B \{ A \}} & \equiv & \langle\rangle \Leftarrow B & \rightarrow & R \Leftarrow A \\ \texttt{class B import A} & \equiv & R \Leftarrow B & \rightarrow & R \Leftarrow A \end{array}$$

Our concrete language of course supports importing/exporting only selected identifiers, and renaming on import/export, but that is beyond the scope of this article. To get an intuitive understanding of how the concrete language is mapped to the core, let's look at the following example:

```
class C1                class C2                   class C
declare ?X              declare ?Y                 import C1 C2
export X                export Y                   {
{                       {                            <syn>
  <syn>                   <syn>                      {X->Y}
  {node X[cat=s]}         {node Y[tense=past]}     }
}                       }
```

`C1` (resp. `C2`) declares local identifier `X` (resp. `Y`) and exports it. Both of these classes accumulate some syntactic descriptions (a new node with some features). `C` imports both these classes and therefore can access `X` and `Y` as if they were locally defined, and adds the syntactic constraint that `X` immediately dominates `Y`. This code gets translated into the core as follows:

$$\begin{array}{rcl} \langle \texttt{X}{:}X\rangle \Leftarrow C_1 & \rightarrow & \quad \mathbf{syn} \mathrel{+}= \mathsf{node}(X) \\ & & \wedge\ \mathbf{syn} \mathrel{+}= X[\texttt{cat} = \texttt{s}] \\ \langle \texttt{Y}{:}Y\rangle \Leftarrow C_2 & \rightarrow & \quad \mathbf{syn} \mathrel{+}= \mathsf{node}(Y) \\ & & \wedge\ \mathbf{syn} \mathrel{+}= Y[\texttt{tense} = \texttt{past}] \\ \langle \texttt{X}{:}X, \texttt{Y}{:}Y\rangle \Leftarrow C & \rightarrow & \quad E_1 \Leftarrow C_1 \ \wedge\ X = E_1.\texttt{X} \\ & & \wedge\ E_2 \Leftarrow C_2 \ \wedge\ Y = E_2.\texttt{Y} \\ & & \wedge\ \mathbf{syn} \mathrel{+}= X \rightarrow Y \end{array}$$

Valuations.

While a grammar traditionally stipulates a start symbol, we have found it more convenient to let the grammar writer supply any number of statements of the form `value` E. For each one, all valuations of E, computed with our non-deterministic MG, are to be contributed to the lexicon.

6 Implementation of the Metagrammar Processor

The processor consists of 3 modules: a front-end to compile the object-oriented concrete syntax into the logical core, a virtual machine (VM) to execute core programs, and a solver to take the resulting accumulated trees descriptions and compute their minimal models, i.e. the TAG trees which they describe.

6.1 Compiler front-end

The compilation process converts the MG object-oriented concrete syntax into our logic programming core, then compiles the latter into instructions for a VM inspired by the Warren Abstract Machine (WAM) [14].

Parsing was implemented using GUMP. The next step of compilation is to take care of scope management and resolve all identifiers. By examining and following import/export declarations, we compute for each class (1) all the identifiers in its scope, (2) its export record. This is sufficient to permit translation into the core.

We then compile the logical core into symbolic code (SCODE) for our VM. Every instruction is represented by a record and we have instructions for conjunction `conj(_ _)` and disjunction `disj(_ _)`.

6.2 An object-oriented virtual machine

The VM implements a fairly standard logic programming kernel with chronological backtracking, but with some extensions. Contrary to the WAM which uses *structure copying*, our VM uses *structure sharing* where a term is represented by a pair of a pattern and an environment in which to interpret it. This technique enables us to save memory space, although pointer dereferencing can be time consuming. The VM is implemented as an object with methods for each instruction: in this manner it can directly execute SCODE. It maintains a stack of instructions (the success continuation), and a trail (the failure continuation) to undo bindings and explore alternatives.

The VM is meant to be extended with support for multiple accumulators. Each extension provides dedicated registers and specialized instructions for accumulating descriptions.

There are a number of reasons why it was more convenient to build our own VM rather than target an existing logic programming language. (1) this makes it easy to extend the VM with efficient support for non-standard datatypes such as open feature structures, properties, nodes and tree descriptions. (2) non-standard datatypes often require non-standard extensions of unification (e.g. the polarities of interaction grammars). (3) advanced constraint programming support is required to compute solutions of accumulated tree descriptions

When the VM has computed a complete derivation for a *valuation* statement, it takes a snapshot of its accumulators and sends it for further processing by the solver. It then backtracks to enumerate all possible derivations.

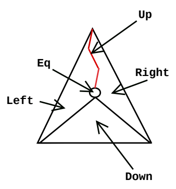
At the end of the execution we possibly have tree descriptions for each valuation of class. For TAG formalism trees are needed, thus we then have to find all the trees that are specifications of those descriptions. Because of the high complexity of this satisfiability problem, we chose a constraint-based approach to decrease the search space.

6.3 A constraint-based tree description solver

In the last stage of processing, the snapshot $(D_1, \ldots, D_n)^3$ taken by the VM is then submitted to a solver module, where, for each dimension i, there is a specialized solver S_i for computing the solutions (models) $S_i(D_i)$ of the corresponding accumulated description D_i. The lexical entries contributed by the snapshot are then: $\{(M_1, \ldots, M_n) \mid M_i \in S_i(D_i) \text{ for } 1 \leq i \leq n\}$

In the case of semantics, the solver is trivial and basically just returns the description itself. However, for syntax, we use a dominance constraint solver based on the set constraint approach of [15] which we extended to implement Crabbé's semantics for the color annotation of nodes.

When observed from a specific node x, the nodes of a solution tree (a model), and hence the variables which they interpret, are partitioned into 5 regions: the node denoted by x itself, all nodes below, all nodes above, all nodes to the left, and all nodes to the right. The main idea is to introduce corresponding set variables Eq_x, Up_x, $Down_x$, $Left_x$, $Right_x$ to encode the sets of variables that are interpreted by nodes in the model which are respectively equal, above, below, left, and right of the node interpreting x. The interested reader should refer to [15] for the precise formalization.



Color constraints.

An innovative aspect of Crabbé's approach is that nodes are decorated with colors (red, black, white) that constrains how they can be merged when computing models. The color combination rules are summarized in the table in the margin: a red node cannot merge with any node, a black node can only merge with white nodes, and a white node must merge with a black node. Thus, in a valid model, we only have red and black nodes; in fact, exactly those which where already present in the input description. Intuitively, black

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

^{3} assuming n dimensions

nodes represent nodes that can be combined, red nodes are nodes that cannot, and white nodes those that must be combined. Thus, in valid models, all white nodes are absorbed by black nodes.

We extend the formalization of [15] with variables RB_x representing the unique red or black node that each x is identified with. We write V_{B}, V_{R}, and V_{W} for the sets of resp. black, red and white variables in the description. A red node cannot be merged with any other node (6), a black node can only be merged with white nodes (7), a white node must be merged with a black node (8):

$$x \in V_{\mathrm{R}} \quad\Rightarrow\quad RB_x = x \quad\wedge\quad Eq_x = \{x\} \tag{6}$$

$$x \in V_{\mathrm{B}} \quad\Rightarrow\quad RB_x = x \tag{7}$$

$$x \in V_{\mathrm{W}} \quad\Rightarrow\quad RB_x \in V_{\mathrm{B}} \tag{8}$$

Finally, two nodes are identified iff they are both identified with the same red or black node. Thus we must extend the clause of [15] for $x \neg{=}\, y$ as follows, where $\|$ denotes disjointness:

$$x \neg{=}\, y \qquad\equiv\qquad (Eq_x \| Eq_y \quad\wedge\quad RB_x \neq RB_y) \tag{9}$$

7 Conclusion

We motivated and presented a metagrammar formalism that embraces a multi-paradigm perspective, and we outlined its implementation in a Mozart-based tool. Our approach is innovative in that it combines an object-oriented management of linguistic abstraction, with a logic programming core to express and enumerate alternatives, and with constraint solving of dominance-based tree descriptions. That is why we chose Mozart/Oz: this multi-paradigm language provides parsing tools along with useful libraries for dealing with constraints.

Our new MG processor has already been used to develop a significant TAG for French, with over 3000 trees. And we are currently interfacing this tool with two parsers: the LORIA LTAG PARSER[4] version 2 [16] and the DyALog[5] system [17]. We are also extending it to support Interaction Grammars [12].

[4] `http://www.loria.fr/~azim/LLP2/help/fr/index.html`
[5] `ftp://ftp.inria.fr/INRIA/Projects/Atoll/Eric.Clergerie/DyALog/`

References

[1] XTAG-Research-Group: A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania (2001) Available at http://www.cis.upenn.edu/~xtag/gramrelease.html.

[2] Abeillé, A., Candito, M., Kinyon, A.: Ftag: current status and parsing scheme. In: VEXTAL, Venice, Italy. (1999)

[3] Kroch, A., Joshi, A.: The linguistic relevance of tree adjoining grammars. Technical report, MS-CIS-85-16, University of Pennsylvania, Philadelphia (1985)

[4] Joshi, A., Schabes, Y.: Tree-adjoining grammars. In Rozenberg, G., Salomaa, A., eds.: Handbook of Formal Languages. Volume 3. Springer, Berlin, New York (1997) 69 – 124

[5] Vijay-Shanker, K., Schabes, Y.: Structure sharing in lexicalized tree adjoining grammars. In: Proceedings of the 16th International Conference on Computational Linguistics (COLING'92), Nantes, pp. 205 - 212. (1992)

[6] Candito, M.: Représentation modulaire et paramétrable de grammaires électroniques lexicalisées : application au français et à l'italien. PhD thesis, Université Paris 7 (1999)

[7] Gaiffe, B., Crabbé, B., Roussanaly, A.: A new metagrammar compiler. In: Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6), Venice. (2002)

[8] Xia, F., Palmer, M., Vijay-Shanker, K.: Toward semi-automating grammar development. In: Proc. of the 5th Natural Language Processing Pacific Rim Symposium(NLPRS-99), Beijing, China. (1999)

[9] Rogers, J., Vijay-Shanker, K.: Reasoning with descriptions of trees. In: Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics, pp. 72 - 80. (1992)

[10] Crabbé, B.: Lexical classes for structuring the lexicon of a tag. In: Proceedings of the Lorraine/Saarland workshop on Prospects and Advances in the Syntax/Semantics Interface. (2003)

[11] Crabbé, B.: Alternations, monotonicity and the lexicon : an application to factorising information in a tree adjoining grammar. In: Proceedings of the 15th ESSLLI, Vienne. (2003)

[12] Perrier, G.: Interaction grammars. In: Proceedings of the 18th International Conference on Computational Linguistics (COLING'2000), Saarbrucken, pp. 600 - 606. (2000)

[13] Van Roy, P.: Extended dcg notation: A tool for applicative programming in prolog. Technical report, Technical Report UCB/CSD 90/583, Computer Science Division, UC Berkeley (1990)

[14] Ait-Kaci, H.: Warren's abstract machine: A tutorial reconstruction. In Furukawa, K., ed.: Logic Programming: Proc. of the Eighth International Conference. MIT Press, Cambridge, MA (1991) 939

[15] Duchier, D.: Constraint programming for natural language processing (2000) Lecture Notes, ESSLLI 2000. Available at http://www.ps.uni-sb.de/Papers/abstracts/duchier-esslli2000.html.

[16] Crabbé, B., Gaiffe, B., Roussanaly, A.: Représentation et gestion du lexique d'une grammaire d'arbres adjoints (2004) Traitement Automatique des Langues, 43,3.

[17] Villemonte de la Clergerie, E.: Designing efficient parsers with DyALog (2004) Slides presented at GLINT, Universidade Nova de Lisboa.