# SemTAG, the LORIA toolbox for TAG-based Parsing and Generation

**Eric Kow**
INRIA / LORIA
Université Henri Poincaré
615, rue du Jardin Botanique
F-54 600 Villers-Lès-Nancy
kow@loria.fr

**Yannick Parmentier**
INRIA / LORIA
Université Henri Poincaré
615, rue du Jardin Botanique
F-54 600 Villers-Lès-Nancy
parmenti@loria.fr

**Claire Gardent**
CNRS / LORIA
615, rue du Jardin Botanique
F-54 600 Villers-Lès-Nancy
gardent@loria.fr

## Abstract

In this paper, we introduce SEMTAG, a toolbox for TAG-based parsing and generation. This environment supports the development of wide-coverage grammars and differs from existing environments for TAG such as XTAG, (XTAG-Research-Group, 2001) in that it includes a semantic dimension. SEMTAG is open-source and freely available.

## 1 Introduction

In this paper we introduce a toolbox that allows for both parsing and generation with TAG. This toolbox combines existing software and aims at facilitating grammar development, More precisely, this toolbox includes[1]:

- XMG: a grammar compiler which supports the generation of a TAG from a factorised TAG (Crabbé and Duchier, 2004),
- LLP2 and DyALog: two chart parsers, one with a friendly user interface (Lopez, 2000) and the other optimised for efficient parsing (Villemonte de la Clergerie, 2005)[2]
- GenI: a chart generator which has been tested on a middle size grammar for French (Gardent and Kow, 2005)

[1] All these tools are freely available, more information and links at http://trac.loria.fr/~semtag.

[2] Note that DyALog refers in fact to a logic programming language, and a tabular compiler for this language. The DyALog system is well-adapted to the compilation of efficient tabular parsers.

## 2 XMG, a grammar writing environment for Tree Based Grammars

XMG provides a grammar writing environment for tree based grammars[3] with three distinctive features. First, XMG supports a highly factorised and fully declarative description of tree based grammars. Second, XMG permits the integration in a TAG of a semantic dimension. Third, XMG is based on well understood and efficient logic programming techniques. Moreover, it offers a graphical interface for exploring the resulting grammar (see Figure 1).

**Factorising information.** In the XMG framework, a TAG is defined by a set of classes organised in an inheritance hierarchy where classes define tree fragments (using a tree logic) and tree fragment combinations (by conjunction or disjunction). XMG furthermore integrates a sophisticated treatment of names whereby variables scope can be local, global or user defined (i.e., local to part of the hierarchy).

In practice, the resulting framework supports a very high degree of factorisation. For instance, a first core grammar (FRAG) for French comprising 4 200 trees was produced from roughly 300 XMG classes.

**Integrating semantic information.** In XMG, classes can be multi-dimensional. That is, they can be used to describe several levels of linguistic knowledge such as for instance, syntax, semantics or prosody. At present, XMG supports classes including both a syntactic and a semantic dimension. As mentioned above, the syntactic dimen-

[3] Although in this paper we only mention TAG, the XMG framework is also used to develop so called Interaction Grammars i.e., grammars whose basic units are tree descriptions rather than trees (Parmentier and Le Roux, 2005).

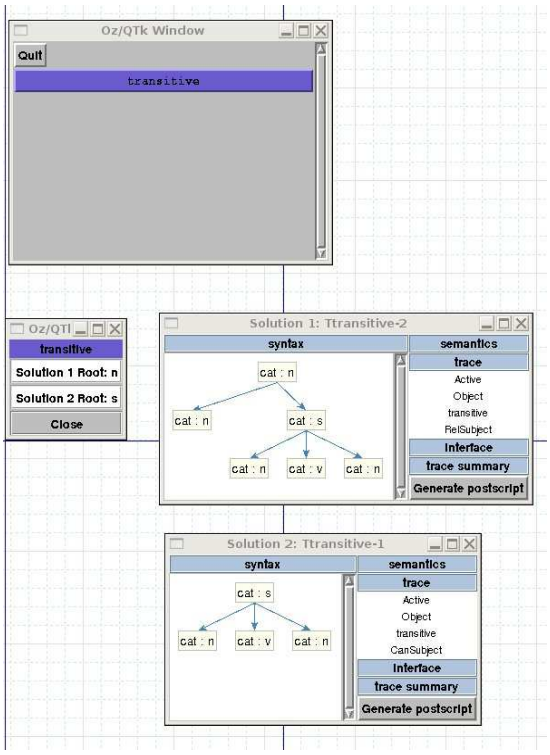


Figure 1: XMG's graphical interface

sion is based on a tree logic and can be used to describe (partial) tree fragments. The semantic dimension on the other hand, can be used to associate with each tree a flat semantic formula. Such a formula can furthermore include identifiers which corefer with identifiers occurring in the associated syntactic tree. In other words, XMG also provides support for the interface between semantic formulae and tree decorations. Note that the inclusion of semantic information remains optional. That is, it is possible to use XMG to define a purely syntactic TAG.

XMG was used to develop a core grammar for French (FRAG) which was evaluated to have 75% coverage[4] on the Test Suite for Natural Language Processing (TSNLP, (Lehmann et al., 1996)). The FRAG grammar was furthermore enriched with semantic information using another 50 classes describing the semantic dimension (Gardent, 2006). The resulting grammar (SEMFRAG) describes both the syntax and the semantics of the French core constructions.

**Compiling an XMG specification.** By building on efficient techniques from logic programming and in particular, on the Warren's Abstract

[4]This means that for 75 % of the sentences, a TAG parser can build at least one derivation.

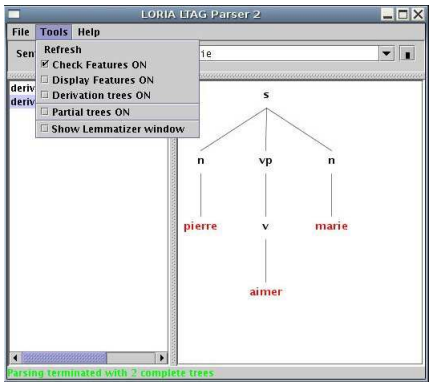


Figure 2: The LLP2 parser.

Machine idea (Ait-Kaci, 1991), the XMG compiler allows for very reasonable compilation times (Duchier et al., 2004). For instance, the compilation of a TAG containing 6 000 trees takes about 15 minutes with a Pentium 4 processor 2.6 GHz and 1 GB of RAM.

## 3 Two TAG parsers

The toolbox includes two parsing systems: the LLP2 parser and the DyALog system. Both of them can be used in conjunction with XMG. First we will briefly introduce both of them, and then show that they can be used with a semantic grammar (e.g., SEMFRAG) to perform not only syntactic parsing but also semantic construction.

**LLP2** The LLP2 parser is based on a bottom-up algorithm described in (Lopez, 1999). It has relatively high parsing times but provides a user friendly graphical parsing environment with much statistical information (see Figure 2). It is well suited for teaching or for small scale projects.

**DyALog** The DyALog system on the other hand, is a highly optimised parsing system based on tabulation and automata techniques (Villemonte de la Clergerie, 2005). It is implemented using the DyALog programming language (*i.e.*, it is bootstrapped) and is also used to compile parsers for other types of grammars such as *Tree Insertion Grammars*.

The DyALog system is coupled with a semantic construction module whose aim is to associate with each parsed string a semantic representation[5]. This module assumes a TAG of the type described in (Gardent and Kallmeyer, 2003; Gardent, 2006)

[5]The corresponding system is called SemConst (*cf* section 6).

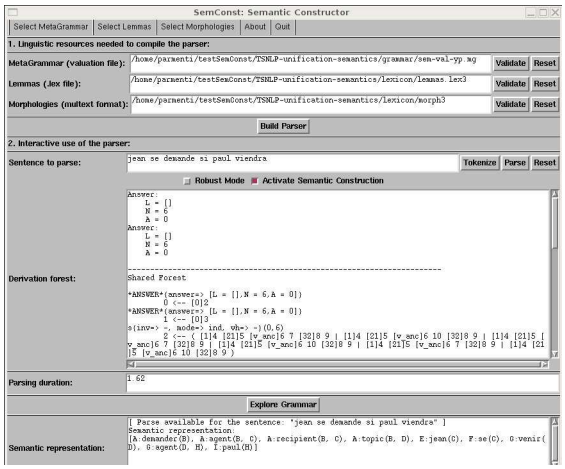


Figure 3: The SemConst system

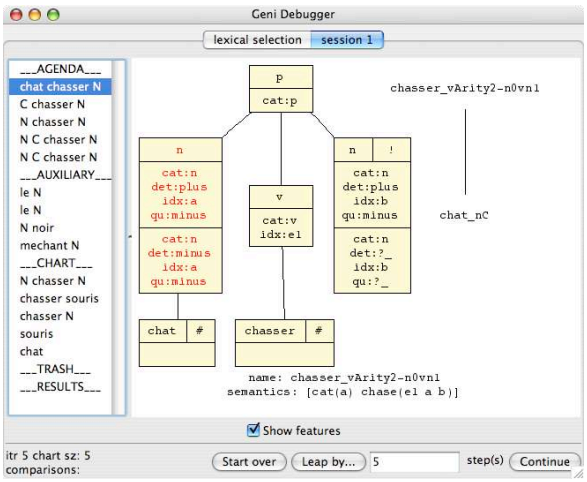


Figure 4: The GenI debugger

where initial trees are associated with semantic information and unification is used to combine semantic representations. In such a grammar, the semantic representation of a derived tree is the union of the semantic representations of the trees entering in the derivation of that derived tree modulo the unifications entailed by analysis. As detailed in (Gardent and Parmentier, 2005), such grammars support two strategies for semantic construction.

The first possible strategy is to use the full grammar and to perform semantic construction during derivation. In this case the parser must manipulate both syntactic trees and semantic representations. The advantage is that the approach is simple (the semantic representations can simply be an added feature on the anchor node of each tree). The drawback is that the presence of semantic information might reduce chart sharing.

The second possibility involves extracting the semantic information contained in the grammar and storing it into a semantic lexicon. Parsing then proceeds with a purely syntactic grammar and semantic construction is done after parsing on the basis of the parser output and of the extracted semantic lexicon. This latter technique is more suitable for large scale semantic construction as it supports better sharing in the derivation forests. It is implemented in the LORIA toolbox where a module permits both extracting a semantic lexicon from a semantic TAG and constructing a semantic representation based on this lexicon and on the derivation forests output by DyALog (see Figure 3).

The integration of the DyALog system into the toolbox is relatively new so that parsing evaluation is still under progress. So far, evaluation has been restricted to parsing the TSNLP with DyALog with the following preliminary results. On sentences ranging from 1 to 18 words, with an average of 7 words per sentence, and with a grammar containing 5 069 trees, DyALog average parsing time is of 0.38 sec with a P4 processor 2.6 GHz and 1 GB of RAM[6].

## 4 A TAG-based surface realiser

The surface realiser GenI takes a TAG and a flat semantic logical form as input, and produces all the sentences that are associated with that logical form by the grammar. It implements two bottom up algorithms, one which manipulates derived trees as items and one which is based on Earley for TAG. Both of these algorithms integrate a number of optimisations such as delayed adjunction and polarity filtering (Kow, 2005; Gardent and Kow, 2005).

GenI is written in Haskell and includes a graphical debugger to inspect the state of the generator at any point in the surface realisation process (see Figure 4). It also integrates a test harness for automated regression testing and benchmarking of the surface realiser and the grammar. The harness gtester is written in Python. It runs the surface realiser on a test suite, outputting a single document with a table of passes and failures and various performance charts (see Figures 5 and 6).

**Test suite and performance** The test suite is built with an emphasis on testing the surface re-

[6]These features only concern classic syntactic parsing as the semantic construction module has not been tested on real grammars yet.

| test | expected | simple | earley |
|---|---|---|---|
| t1 | il le accepter | pass | pass |
| t32 | il nous accepter | pass | pass |
| t83 | le ingnieur le lui apprendre | pass | DIED |
| t114 | le ingnieur nous le prsenter | pass | pass |
| t145 | le ingnieur vous le apprendre | pass | pass |
| t180 | vous venir | pass | pass |

Figure 5: Fragment of test harness output - The Earley algorithm timed out.

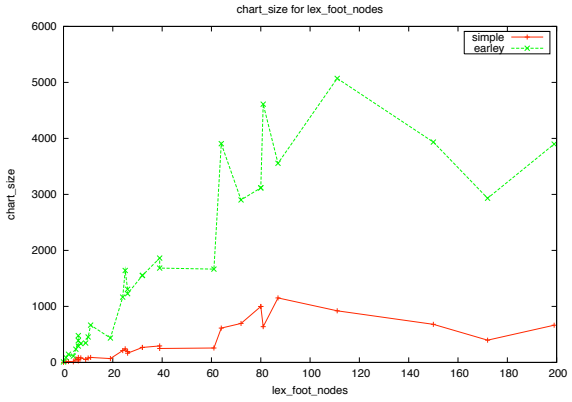


Figure 6: Automatically generated graph of performance data by the test harness.

aliser's performance in the face of increasing paraphrastic power i.e., ambiguity. The suite consists of semantic inputs that select for and combines verbs with different valencies. For example, given a hypothetical English grammar, a valency (2,1) semantics might be realised in as *Martin thinks Faye drinks* (*thinks* takes 2 arguments and *drinks* takes 1), whereas a valency (2,3,2) one would be *Dora says that Martin tells Bob that Faye likes music*. The suite also adds a varying number of intersective modifiers into the mix, giving us for instance, *The girl likes music*, *The pretty scary girl likes indie music*.

The sentences in the suite range from 2 to 15 words (8 average). Realisation times for the core suite range from 0.7 to 2.84 seconds CPU time (average 1.6 seconds).

We estimate the ambiguity for each test case in two ways. The first is to count the number of paraphrases. Given our current grammar, the test cases in our suite have up to 669 paraphrases (average 41). The second estimate for ambiguity is the number of combinations of lexical items covering the input semantics.

This second measure is based on optimisation known as polarity filtering (Gardent and Kow, 2005). This optimisation detects and eliminates combinations of lexical items that cannot be used to build a result. It associates the syntactic resources (root nodes) and requirements (substitution nodes) of the lexical items to polarities, which are then used to build "polarity automata". The automata are minimised to eliminate lexical combinations where the polarities do not cancel out, that is those for which the number of root and substitution nodes for any given category do not equal each other.

Once built, the polarity automata can also serve to estimate ambiguity. The number of paths in the automaton represent the number of possible combinations of lexical items. To determine how effective polarity filtering with respect to ambiguity, we compare the combinations before and after polarity filtering. Before filtering, we start with an initial polarity automaton in which all items are associated with a zero polarity. This gives us the lexical ambiguity before filtering. The polarity filter then builds upon this to form a final automaton where all polarities are taken into account. Counting the paths on this automaton gives us the ambiguity after filtering, and comparing this number with the lexical initial ambiguity provides an estimate on the usefulness of the polarity filter. In our suite, the initial automata for each case have 1 to 800 000 paths (76 000 average). The final automata have 1 to 6000 paths (192 average). This can represent quite a large reduction in search space, 4000 times in the case of the largest automaton. The effect of this search space reduction is most pronounced on the larger sentences or those with the most modifiers. Indeed, realisation times with and without filtering are comparable for most of the test suite, but for the most complicated sentence in the core suite, polarity filtering makes surface realisation 94% faster, producing a result in 2.35 seconds instead of 37.38.

## 5 Benefits of an integrated toolset

As described above, the LORIA toolbox for TAG based semantic processing includes a lexicon, a grammar, a parser, a semantic construction module and a surface realiser. Integrating these into a single platform provides some accrued benefits which we now discuss in more details.

**Simplified resource management** The first advantage of an integrated toolkit is that it facilitates

the management of the linguistic resources used namely the grammar and the lexicon. Indeed it is common that each NLP tool (parser or generator) has its own representation format. Thus, managing the resources gets tiresome as one has to deal with several versions of a single resource. When one version is updated, the others have to be recomputed. Using an integrated toolset avoid such a drawback as the intermediate formats are hidden and the user can focus on linguistic description.

**Better support for grammar development** When developing parsers or surface realisers, it is useful to test them out by running them on large, realistic grammars. Such grammars can explore nooks and crannies in our implementations that would otherwise have been overlooked by a toy grammar. For example, it was only when we ran `GenI` on our French grammar that we realised our implementation did not account for auxiliary trees with substitution nodes (this has been rectified). In this respect, one could argue that `XMG` could almost be seen as a parser/realiser debugging utility because it helps us to build and extend the large grammars that are crucial for testing.

This perspective can also be inverted; parsers and surface realiser make for excellent grammar-debugging devices. For example, one possible regression test is to run the parser on a suite of known sentences to make sure that the modified grammar still parses them correctly. The exact reverse is useful as well; we could also run the surface realiser over a suite of known semantic inputs and make sure that sentences are generated for each one. This is useful for two reasons. First, reading surface realiser output (sentences) is arguably easier for human beings than reading parser output (semantic formulas). Second, the surface realiser can tell us if the grammar overgenerates because it would output nonsense sentences. Parsers, on the other hand, are much better adapted for testing for undergeneration because it is easier to write sentences than semantic formulas, which makes it easier to test phenomena which might not already be in the suite.

**Towards a reversible grammar** Another advantage of using such a toolset relies on the fact that we can manage a common resource for both parsing and generation, and thus avoid inconsistency, redundancy and offer a better flexibility as advocated in (Neumann, 1994).

On top of these practical questions, having a unique reversible resource can lead us further. For instance, (Neumann, 1994) proposes an interleaved parsing/realisation architecture where the parser is used to choose among a set of paraphrases proposed by the generator; paraphrases which are ambiguous (that have multiple parses) are discarded in favour of those whose meaning is most explicit. Concretely, we could do this with a simple pipeline using `GenI` to produce the paraphrases, `DyALog` to parse them, and a small shell script to pick the best result. This would only be a simulation, of course. (Neumann, 1994) goes as far as to interleave the processes, keeping the shared chart and using the parser to iteratively prune the search space as it is being explored by the generator. The version we propose would not have such niceties as a shared chart, but the point is that having all the tools at our disposable makes such experimentation possible in the first place.

Moreover, there are several other interesting applications of the combined toolbox. We could use the surface realiser to build artificial corpora. These can in turn be parsed to semi-automatically create rich treebanks containing syntactico-semantic analyses à la Redwoods (Oepen et al., 2002).

Eventually, another use for the toolbox might be in components of standard NLP applications such as machine translation, questioning answering, or interactive dialogue systems.

## 6 Availability

The toolbox presented here is open-source and freely available under the terms of the GPL[7]. More information about the requirements and installation procedure is available at `http://trac.loria.fr/~semtag`. Note that this toolbox is made of two main components: the GenI[8] system and the SemConst[9] system, which respectively performs generation and parsing from common linguistic resources. The first is written in Haskell (except the `XMG` part written in Oz) and is multi-platform (Linux, Windows, Mac OS). The latter is written in Oz (except the `DyALog` part which is bootstrapped and contains some Intel assembler code) and is available on Unix-like plat-

[7] Note that `XMG` is released under the terms of the CeCILL license (`http://www.cecill.info/index.en.html`), which is compatible with the GPL.

[8] `http://trac.loria.fr/~geni`

[9] `http://trac.loria.fr/~semconst`

forms only.

## 7 Conclusion

The LORIA toolbox provides an integrated environment for TAG based semantic processing: either to construct the semantic representation of a given sentence (parsing) or to generate a sentence verbalising a given semantic content (generation).

Importantly, both the generator and the parsers use the same grammar (SEMFRAG) so that both tools can be used jointly to improve grammar precision. All the sentences outputted by the surface realiser should be parsed to have at least the semantic representation given by the test suite, and all parses of a sentence should be realised into at least the same sentence.

Current and future work concentrates on developing an automated error mining environment for both parsing and generation; on extending the grammar coverage; on integrating further optimisations both in the parser (through parsing with factorised trees) and in the generator (through packing and accessibility filtering cf. (Carroll and Oepen, 2005); and on experimenting with different semantic construction strategies (Gardent and Parmentier, 2005).

## References


H. Ait-Kaci. 1991. Warren's Abstract Machine: A Tutorial Reconstruction. In K. Furukawa, editor, *Proc. of the Eighth International Conference of Logic Programming*. MIT Press, Cambridge, MA.

J. Carroll and S. Oepen. 2005. High efficiency realization for a wide-coverage unification grammar. In R. Dale and K-F. Wong, editors, *Proceedings of the Second International Joint Conference on Natural Language Processing*, volume 3651 of *Springer Lecture Notes in Artificial Intelligence*, pages 165–176.

B. Crabbé and D. Duchier. 2004. Metagrammar Redux. In *Proceedings of CSLP 2004, Copenhagen*.

D. Duchier, J. Le Roux, and Y. Parmentier. 2004. The Metagrammar Compiler: An NLP Application with a Multi-paradigm Architecture. In *2nd International Mozart/Oz Conference (MOZ'2004)*, Charleroi.

C. Gardent and L. Kallmeyer. 2003. Semantic construction in FTAG. In *Proceedings of EACL'03, Budapest*.

C. Gardent and E. Kow. 2005. Generating and selecting grammatical paraphrases. *ENLG, Aberdeen*.

C. Gardent and Y. Parmentier. 2005. Large scale semantic construction for tree adjoining grammars. In *Proceedings of The Fifth International Conference on Logical Aspects of Computational Linguistics (LACL05)*.

C. Gardent. 2006. Intégration d'une dimension sémantique dans les grammaires d'arbres adjoints. In *Actes de la conférence TALN'2006 Leuven*.

E. Kow. 2005. Adapting polarised disambiguation to surface realisation. In *17th European Summer School in Logic, Language and Information - ESSLLI'05, Edinburgh, UK*, Aug.

S. Lehmann, S. Oepen, S. Regnier-Prost, K. Netter, V. Lux, J. Klein, K. Falkedal, F. Fouvry, D. Estival, E. Dauphin, H. Compagnion, J. Baur, L. Balkan, and D. Arnold. 1996. TSNLP — Test Suites for Natural Language Processing. In *Proceedings of COLING 1996*, Kopenhagen.

P. Lopez. 1999. *Analyse d'énoncés oraux pour le dialogue homme-machine à l'aide de grammaires lexicalisées d'arbres*. Ph.D. thesis, Université Henri Poincaré – Nancy 1.

P. Lopez. 2000. Extended Partial Parsing for Lexicalized Tree Grammars. In *Proceedings of the International Workshop on Parsing Technology (IWPT2000), Trento, Italy.*

G. Neumann. 1994. *A Uniform Computational Model for Natural Language Parsing and Generation*. Ph.D. thesis, University of the Saarland, Saarbrücken.

S. Oepen, E. Callahan, C. Manning, and K. Toutanova. 2002. Lingo redwoods—a rich and dynamic treebank for hpsg.

Y. Parmentier and J. Le Roux. 2005. XMG: an Extensible Metagrammatical Framework. In *Proceedings of the Student Session of the 17th European Summer School in Logic, Language and Information, Edinburg, Great Britain*, Aug.

E. Villemonte de la Clergerie. 2005. DyALog: a tabular logic programming based environment for NLP. In *Proceedings of CSLP'05*, Barcelona.

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