Generating XTAG Parsers from Algebraic Specifications*

Carlos Gómez-Rodríguez and **Miguel A. Alonso**
Departamento de Computación
Universidade da Coruña
Campus de Elviña, s/n
15071 A Coruña, Spain
{cgomezr, alonso}@udc.es

Manuel Vilares
E. S. de Ingeniería Informática
Universidad de Vigo
Campus As Lagoas, s/n
32004 Ourense, Spain
vilares@uvigo.es

Abstract

In this paper, a generic system that generates parsers from parsing schemata is applied to the particular case of the XTAG English grammar. In order to be able to generate XTAG parsers, some transformations are made to the grammar, and TAG parsing schemata are extended with feature structure unification support and a simple tree filtering mechanism. The generated implementations allow us to study the performance of different TAG parsers when working with a large-scale, wide-coverage grammar.

1 Introduction

Since Tree Adjoining Grammars (TAG) were introduced, several different parsing algorithms for these grammars have been developed, each with its peculiar characteristics. Identifying the advantages and disadvantages of each of them is not trivial, and there are no comparative studies between them in the literature that work with real-life, wide coverage grammars. In this paper, we use a generic tool based on parsing schemata to generate implementations of several TAG parsers and compare them by parsing with the XTAG English Grammar (XTAG, 2001).

The parsing schemata formalism (Sikkel, 1997) is a framework that allows us to describe parsers in a simple and declarative way. A parsing schema is a representation of a parsing algorithm as a set of inference rules which are used to perform deductions on intermediate results called items. These items represent sets of incomplete parse trees which the algorithm can generate. An input sentence to be analyzed produces an initial set of items. Additionally, a parsing schema must define a criterion to determine which items are final, i.e. which items correspond to complete parses of the input sentence. If it is possible to obtain a final item from the set of initial items by using the schema's inference rules (called deductive steps), then the input sentence belongs to the language defined by the grammar. The parse forest can then be retrieved from the intermediate items used to infer the final items, as in (Billot and Lang, 1989).

As an example, we introduce a CYK-based algorithm (Vijay-Shanker and Joshi, 1985) for TAG. Given a tree adjoining grammar $G = (V_T, V_N, S, I, A)$[1] and a sentence of length n which we denote by $a_1\ a_2\ \ldots\ a_n$[2], we denote by $P(G)$ the set of productions $\{N^\gamma \rightarrow N_1^\gamma N_2^\gamma \ldots N_r^\gamma\}$ such that N^γ is an inner node of a tree $\gamma \in (I \cup A)$, and $N_1^\gamma N_2^\gamma \ldots N_r^\gamma$ is the ordered sequence of direct children of N^γ.

The parsing schema for the TAG CYK-based algorithm (Alonso et al., 1999) is a function that maps such a grammar G to a deduction system whose domain is the set of items

$\{[N^\gamma, i, j, p, q, adj]\}$

verifying that N^γ is a tree node in an elementary

* Partially supported by Ministerio de Educación y Ciencia and FEDER (TIN2004-07246-C03-01, TIN2004-07246-C03-02), Xunta de Galicia (PGIDIT05PXIC30501PN, PGIDIT05PXIC10501PN, PGIDIT05SIN044E and PGIDIT05SIN059E), and Programa de becas FPU (Ministerio de Educación y Ciencia). We are grateful to Eric Villemonte de la Clergerie and François Barthelemy for their help in converting the XTAG grammar to XML.

[1] Where V_T denotes the set of terminal symbols, V_N the set of nonterminal symbols, S the axiom, I the set of initial trees and A the set of auxiliary trees.

[2] From now on, we will follow the usual conventions by which nonterminal symbols are represented by uppercase letters (A, $B \ldots$), and terminals by lowercase letters (a, $b \ldots$). Greek letters (α, β...) will be used to represent trees, N^γ a node in the tree γ, and R^γ the root node of the tree γ.

tree $\gamma \in (I \cup A)$, i and j ($0 \leq i \leq j$) are string positions, p and q may be undefined or instantiated to positions $i \leq p \leq q \leq j$ (the latter only when $\gamma \in A$), and $adj \in \{true, false\}$ indicates whether an adjunction has been performed on node N^γ.

The positions i and j indicate that a substring $a_{i+1} \ldots a_j$ of the string is being recognized, and positions p and q denote the substring dominated by γ's foot node. The final item set would be

$$\{[R^\alpha, 0, n, -, -, adj] \mid \alpha \in I\}$$

for the presence of such an item would indicate that there exists a valid parse tree with yield $a_1\ a_2 \ldots a_n$ and rooted at R^α, the root of an initial tree; and therefore there exists a complete parse tree for the sentence.

A *deductive step* $\frac{\eta_1 \ldots \eta_m}{\xi}\ \Phi$ allows us to infer the item specified by its consequent ξ from those in its antecedents $\eta_1 \ldots \eta_m$. *Side conditions* (Φ) specify the valid values for the variables appearing in the antecedents and consequent, and may refer to grammar rules or specify other constraints that must be verified in order to infer the consequent. The deductive steps for our CYK-based parser are shown in figure 1. The steps $\mathcal{D}^{\text{Scan}}_{\text{CYK}}$ and $\mathcal{D}^{\epsilon}_{\text{CYK}}$ are used to start the bottom-up parsing process by recognizing a terminal symbol for the input string, or none if we are using a tree with an epsilon node. The $\mathcal{D}^{\text{Binary}}_{\text{CYK}}$ step (where the operation $p \cup p'$ returns p if p is defined, and p' otherwise) represents the bottom-up parsing operation which joins two subtrees into one, and is analogous to one of the deductive steps of the CYK parser for CFG. The $\mathcal{D}^{\text{Unary}}_{\text{CYK}}$ step is used to handle unary branching productions. $\mathcal{D}^{\text{Foot}}_{\text{CYK}}$ and $\mathcal{D}^{\text{Adj}}_{\text{CYK}}$ implement the adjunction operation, where a tree β is adjoined into a node N^γ; their side condition $\beta \in \text{adj}(N^\gamma)$ means that β must be adjoinable into the node N^γ (which involves checking that N^γ is an adjunction node, comparing its label to $\mathbf{R}^\beta$'s and verifying that no adjunction constraint disallows the operation). Finally, the $\mathcal{D}^{\text{Subs}}_{\text{CYK}}$ step implements the substitution operation in grammars supporting it.

As can be seen from the example, parsing schemata are simple, high-level descriptions that convey the fundamental semantics of parsing algorithms while abstracting implementation details: they define a set of possible intermediate results and allowed operations on them, but they don't specify data structures for storing the results or an order for the operations to be executed. This high abstraction level makes schemata useful for defining, comparing and analyzing parsers in pencil and paper without worrying about implementation details. However, if we want to actually execute the parsers and analyze their results and performance in a computer, they must be implemented in a programming language, making it necessary to lose the high level of abstraction in order to obtain functional and efficient implementations.

In order to bridge this gap between theory and practice, we have designed and implemented a system able to automatically transform parsing schemata into efficient Java implementations of their corresponding algorithms. The input to this system is a simple and declarative representation of a parsing schema, which is practically equal to the formal notation that we used previously. For example, this is the $\mathcal{D}^{\text{Binary}}_{\text{CYK}}$ deductive step shown in figure 1 in a format readable by our compiler:

```
@step CYKBinary
[ Node1 , i , k , p , q , adj1 ]
[ Node2 , k , j , p' , q' , adj2 ]
------------------------------ Node3 -> Node1 Node2
[ Node3 , i , j , Union(p;p') , Union(q;q') , false ]
```

The parsing schemata compilation technique used by our system is based on the following fundamental ideas (Gómez-Rodríguez et al., 2006a):

- Each deductive step is compiled to a Java class containing code to match and search for antecedent items and generate the corresponding conclusions from the consequent.
- The step classes are coordinated by a deductive parsing engine, as the one described in (Shieber et al., 1995). This algorithm ensures a sound and complete deduction process, guaranteeing that all items that can be generated from the initial items will be obtained.
- To attain efficiency, an automatic analysis of the schema is performed in order to create indexes allowing fast access to items. As each different parsing schema needs to perform different searches for antecedent items, the index structures we generate are schema-specific. In this way, we guarantee constant-time access to items so that the computational complexity of our generated implementations is never above the theoretical complexity of the parsers.
- Since parsing schemata have an open notation, for any mathematical object can potentially appear inside items, the system includes an extensibility mechanism which can be used to define new kinds of objects to use in schemata.

$$\mathcal{D}_{\text{CYK}}^{\text{Scan}} = \frac{[a, i, i+1]}{[N^\gamma, i, i+1 \mid -, - \mid \text{false}]} \; a = \text{label}(N^\gamma) \qquad \mathcal{D}_{\text{CYK}}^{\epsilon} = \frac{}{[N^\gamma, i, i \mid -, - \mid \text{false}]} \; \epsilon = \text{label}(N^\gamma)$$

$$\mathcal{D}_{\text{CYK}}^{\text{Unary}} = \frac{[M^\gamma, i, j \mid p, q \mid adj]}{[N^\gamma, i, j \mid p, q] \mid \text{false}]} \quad N^\gamma \rightarrow M^\gamma \in \mathcal{P}(\gamma) \qquad \mathcal{D}_{\text{CYK}}^{\text{Binary}} = \frac{\begin{array}{c}[M^\gamma, i, k \mid p, q \mid adj1], \\ [P^\gamma, k, j \mid p', q' \mid adj2]\end{array}}{[N^\gamma, i, j \mid p \cup p', q \cup q' \mid \text{false}]} \quad N^\gamma \rightarrow M^\gamma P^\gamma \in \mathcal{P}(\gamma)$$

$$\mathcal{D}_{\text{CYK}}^{\text{Foot}} = \frac{[N^\gamma, i, j \mid p, q \mid \text{false}]}{[\mathbf{F}^\beta, i, j \mid i, j \mid \text{false}]} \; \beta \in \text{adj}(N^\gamma) \qquad \mathcal{D}_{\text{CYK}}^{\text{Adj}} = \frac{\begin{array}{c}[\mathbf{R}^\beta, i', j' \mid i, j \mid adj], \\ [N^\gamma, i, j \mid p, q \mid \text{false}]\end{array}}{[N^\gamma, i', j' \mid p, q \mid \text{true}]} \; \beta \in \text{adj}(N^\gamma)$$

$$\mathcal{D}_{\text{CYK}}^{\text{Subs}} = \frac{[\mathbf{R}^\alpha, i, j \mid -, - \mid adj]}{[N^\gamma, i, j \mid -, - \mid \text{false}]} \; \alpha \in \text{subs}(N^\gamma)$$

Figure 1: A CYK-based parser for TAG.

2 Generating parsers for the XTAG grammar

By using parsing schemata as the ones in (Alonso et al., 1999; Nederhof, 1999) as input to our system, we can easily obtain efficient implementations of several TAG parsing algorithms. In this section, we describe how we have dealt with the particular characteristics of the XTAG grammar in order to make it compatible with our generic compilation technique; and we also provide empirical results which allow us to compare the performance of several different TAG parsing algorithms in the practical case of the XTAG grammar. It shall be noted that similar comparisons have been made with smaller grammars, such as simplified subsets of the XTAG grammar, but not with the whole XTAG grammar with all its trees and feature structures. Therefore, our comparison provides valuable information about the behavior of various parsers on a complete, large-scale natural language grammar. This behavior is very different from the one that can be observed on small grammars, since grammar size becomes a dominant factor in computational complexity when large grammars like the XTAG are used to parse relatively small natural language sentences (Gómez-Rodríguez et al., 2006b).

2.1 Grammar conversion

The first step we undertook in order to generate parsers for the XTAG grammar was a full conversion of the grammar to an XML-based format, a variant of the TAG markup language (TAGML). In this way we had the grammar in a well-defined format, easy to parse and modify. During this conversion, the trees' anchor nodes were duplicated in order to make our generic TAG parsers allow adjunctions on anchor nodes, which is allowed in the XTAG grammar.

2.2 Feature structure unification

Two strategies may be used in order to take unification into account in parsing: feature structures can be unified after parsing or during parsing. We have compared the two approaches for the XTAG grammar (see table 1), and the general conclusion is that unification during parsing performs better for most of the sentences, although its runtimes have a larger variance and it performs much worse for some particular cases.

In order to implement unification during parsing in our parsing schemata based system, we must extend our schemata in order to perform unification. This can be done in the following way:

- Items are extended so that they will hold a feature structure in addition to the rest of the information they include.
- We need to define two operations on feature structures: the unification operation and the "keep variables" operation. The "keep variables" operation is a transformation on feature structures that takes a feature structure as an argument, which may contain features, values, symbolic variables and associations between them, and returns a feature structure containing only the variable-value associations related to a given elementary tree, ignoring the variables and values not associated through these relations, and completely ignoring features.
- During the process of parsing, feature structures that refer to the same node, or to nodes that are taking part in a substitution or adjunction and

Strategy	Mean	T. Mean 10%	T. Mean 20%	1st Quart.	Median	3rd Quart.	Std. Dev.	Wilcoxon
During	108,270	12,164	7,812	1,585	4,424	9,671	388,010	0.4545
After	412,793	10,710	10,019	2,123	9,043	19,073	14,235	

Table 1: Runtimes in ms of an Earley-based parser using two different unification strategies: unification during and after parsing. The following data are shown: mean, trimmed means (10 and 20%), quartiles, standard deviation, and p-value for the Wilcoxon paired signed rank test (the p-value of 0.4545 indicates that no statistically significant difference was found between the medians).

are going to collapse to a single node in the final parse tree, must be unified. For this to be done, the test that these nodes must unify is added as a side condition to the steps that must handle them, and the unification results are included in the item generated by the consequent. Of course, considerations about the different role of the top and bottom feature structures in adjunction and substitution must be taken into account when determining which feature structures must be unified.

- Feature structures in items must only hold variable-value associations for the symbolic variables appearing in the tree to which the structures refer, for these relationships hold the information that we need in order to propagate values according to the rules specified in the unification equations. Variable-value associations referring to different elementary trees are irrelevant when parsing a given tree, and feature-value and feature-variable associations are local to a node and can't be extrapolated to other nodes, so we won't propagate any of this information in items. However, it must be used locally for unification. Therefore, steps perform unification by using the information in their antecedent items and recovering complete feature structures associated to nodes directly from the grammar, and then use the "keep-variables" operation to remove the information that we don't need in the consequent item.
- In some algorithms, such as CYK, a single deductive step deals with several different elementary tree nodes that don't collapse into one in the final parse tree. In this case, several "keep variables" operations must be performed on each step execution, one for each of these nodes. If we just unified the information on all the nodes and called "keep variables" at the end, we could propagate information incorrectly.
- In Earley-type algorithms, we must take a decision about how predictor steps handle feature structures. Two options are possible: one is propagating the feature structure in the antecedent item to the consequent, and the other is discarding the feature structure and generating a consequent whose associated feature structure is empty. The first option has the advantage that violations of unification constraints are detected earlier, thus avoiding the generation of some items. However, in scenarios where a predictor is applied to several items differing only in their associated feature structures, this approach generates several different items while the discarding approach collapses them into a single consequent item. Moreover, the propagating approach favors the appearance of items with more complex feature structures, thus making unification operations slower. In practice, for XTAG we have found that these drawbacks of propagating the structures overcome the advantages, especially in complex sentences, where the discarding approach performs much better.

2.3 Tree filtering

The full XTAG English grammar contains thousands of elementary trees, so performance is not good if we use the whole grammar to parse each sentence. Tree selection filters (Schabes and Joshi, 1991) are used to select a subset of the grammar, discarding the trees which are known not to be useful given the words in the input sentence.

To emulate this functionality in our parsing schema-based system, we have used its extensibility mechanism to define a function *Selects-tree(a,T)* that returns $true$ if the terminal symbol *a* selects the tree *T*. The implementation of this function is a Java method that looks for this information in XTAG's syntactic database. Then the function is inserted in a filtering step on our schemata:

$$\frac{[a, i, j]}{[Selected, \alpha]} \; alpha \in Trees / \textsc{Selects-tree}(A;\alpha)$$

The presence of an item of the form $[Selected, \alpha]$ indicates that the tree α has been selected by the filter and can be used for parsing. In order for the filter to take effect, we add $[Selected, \alpha]$ as an antecedent to every step in our schemata introducing a new tree α into the parse (such as initters, substitution and adjoining steps). In this way we guarantee that no trees that don't pass the filter will be used for parsing.

3 Comparing several parsers for the XTAG grammar

In this section, we make a comparison of several different TAG parsing algorithms — the CYK-based algorithm described at (Vijay-Shanker and Joshi, 1985), Earley-based algorithms with (Alonso et al., 1999) and without (Schabes, 1994) the valid prefix property (VPP), and Nederhof's algorithm (Nederhof, 1999) — on the XTAG English grammar (release 2.24.2001), by using our system and the ideas we have explained. The schemata for these algorithms without unification support can be found at (Alonso et al., 1999). These schemata were extended as described in the previous sections, and used as input to our system which generated their corresponding parsers. These parsers were then run on the test sentences shown in table 2, obtaining the performance measures (in terms of runtime and amount of items generated) that can be seen in table 3. Note that the sentences are ordered by minimal runtime.

As we can see, the execution times are not as good as the ones we would obtain if we used Sarkar's XTAG distribution parser written in C (Sarkar, 2000). This is not surprising, since our parsers have been generated by a generic tool without knowledge of the grammar, while the XTAG parser has been designed specifically for optimal performance in this grammar and uses additional information (such as tree usage frequency data from several corpora, see (XTAG, 2001)).

However, our comparison allows us to draw conclusions about which parsing algorithms are better suited for the XTAG grammar. In terms of memory usage, CYK is the clear winner, since it clearly generates less items than the other algorithms, and a CYK item doesn't take up more memory than an Earley item.

On the other hand, if we compare execution times, there is not a single best algorithm, since the performance results depend on the size and complexity of the sentences. The Earley-based algorithm with the VPP is the fastest for the first, "easier" sentences, but CYK gives the best results for the more complex sentences. In the middle of the two, there are some sentences where the best performance is achieved by the variant of Earley that doesn't verify the valid prefix property. Therefore, in practical cases, we should take into account the most likely kind of sentences that will be passed to the parser in order to select the best algorithm.

Nederhof's algorithm is always the one with the slowest execution time, in spite of being an improvement of the VPP Earley parser that reduces worst-case time complexity. This is probably because, when extending the Nederhof schema in order to support feature structure unification, we get a schema that needs more unification operations than Earley's and has to use items that store several feature structures. Nederhof's algorithm would probably perform better in relation to the others if we had used the strategy of parsing without feature structures and then performing unification on the output parse forest.

4 Conclusions

A generic system that generates parsers from algebraic specifications (parsing schemata) has been applied to the particular case of the XTAG grammar. In order to be able to generate XTAG parsers, some transformations were made to the grammar, and TAG parsing schemata were extended with feature structure unification support and a simple tree filtering mechanism.

The generated implementations allow us to compare the performance of different TAG parsers when working with a large-scale grammar, the XTAG English grammar. In this paper, we have shown the results for four algorithms: a CYK-based algorithm, Earley-based algorithms with and without the VPP, and Nederhof's algorithm. The result shows that the CYK-based parser is the least memory-consuming algorithm. By measuring execution time, we find that CYK is the fastest algorithm for the most complex sentences, but the Earley-based algorithm with the VPP is the fastest for simpler cases. Therefore, when choosing a parser for a practical application, we should take

1. He was a cow	9. He wanted to go to the city
2. He loved himself	10. That woman in the city contributed to this article
3. Go to your room	11. That people are not really amateurs at intelectual duelling
4. He is a real man	12. The index is intended to measure future economic performance
5. He was a real man	13. They expect him to cut costs throughout the organization
6. Who was at the door	14. He will continue to place a huge burden on the city workers
7. He loved all cows	15. He could have been simply being a jerk
8. He called up her	16. A few fast food outlets are giving it a try

Table 2: Test sentences.

Sentence	Runtimes in milliseconds				Items generated			
	Parser				Parser			
	CYK	Ear. no VPP	Ear. VPP	Neder.	CYK	Ear. no VPP	Ear. VPP	Neder.
1	2985	**750**	**750**	2719	1341	1463	**1162**	1249
2	3109	1562	**1219**	6421	**1834**	2917	2183	2183
3	4078	1547	**1406**	6828	**2149**	2893	2298	2304
4	4266	1563	**1407**	4703	1864	1979	**1534**	2085
5	4234	1921	**1421**	4766	1855	1979	**1534**	2085
6	4485	1813	**1562**	7782	**2581**	3587	2734	2742
7	5469	2359	**2344**	11469	**2658**	3937	3311	3409
8	7828	4906	**3563**	15532	**4128**	8058	4711	4716
9	10047	4422	**4016**	18969	**4931**	6968	5259	5279
10	13641	**6515**	7172	31828	**6087**	8828	7734	8344
11	16500	**7781**	15235	56265	**7246**	12068	13221	13376
12	16875	17109	**9985**	39132	**7123**	10428	9810	10019
13	25859	**12000**	20828	63641	**10408**	12852	15417	15094
14	54578	**35829**	57422	178875	**20760**	31278	40248	47570
15	**62157**	113532	109062	133515	**22115**	37377	38824	59603
16	**269187**	3122860	3315359		**68778**	152430	173128	

Table 3: Runtimes and amount of items generated by different XTAG parsers on several sentences. The machine used for all the tests was an Intel Pentium 4 / 3.40 GHz, with 1 GB RAM and Sun Java Hotspot virtual machine (version 1.4.2_01-b06) running on Windows XP. Best results for each sentence are shown in boldface.

into account the kinds of sentences most likely to be used as input in order to select the most suitable algorithm.

References

M. A. Alonso, D. Cabrero, E. de la Clergerie, and M. Vilares. 1999. Tabular algorithms for TAG parsing. *Proc. of EACL'99*, pp. 150–157, Bergen, Norway.

S. Billot and B. Lang. 1989. The structure of shared forest in ambiguous parsing. *Proc. of ACL'89*, pp. 143–151, Vancouver, Canada.

C. Gómez-Rodríguez, J. Vilares and M. A. Alonso. 2006. Automatic Generation of Natural Language Parsers from Declarative Specifications. *Proc. of STAIRS 2006*, Riva del Garda, Italy. Long version available at http://www.grupocole.org/GomVilAlo2006a_long.pdf

C. Gómez-Rodríguez, M. A. Alonso and M. Vilares. 2006. On Theoretical and Practical Complexity of TAG Parsers. *Proc. of Formal Grammars 2006*, Malaga, Spain.

M.-J. Nederhof. 1999. The computational complexity of the correct-prefix property for TAGs. *Computational Linguistics*, 25(3):345–360.

A. Sarkar. 2000. Practical experiments in parsing using tree adjoining grammars. *Proc. of TAG+5*, Paris.

Y. Schabes and A. K. Joshi. 1991. Parsing with lexicalized tree adjoining grammar. In Masaru Tomita, editor, *Current Issues in Parsing Technologies*, pp. 25–47. Kluwer Academic Publishers, Norwell.

Y. Schabes. 1994. Left to right parsing of lexicalized tree-adjoining grammars. *Computational Intelligence*, 10(4):506–515.

S. M. Shieber, Y. Schabes, and F. C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36.

K. Sikkel. 1997. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Springer-Verlag, Berlin.

K. Vijay-Shanker and A. K. Joshi. 1985. Some computational properties of tree adjoining grammars. *Proc. of ACL'85*, pp. 82–93, Chicago, USA.

XTAG Research Group. 2001. A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania.