Prototyping Efficient Natural Language Parsers

Carlos Gómez-Rodríguez, Jesús Vilares and Miguel A. Alonso
Universidade da Coruña
Campus de Elviña s/n
15071 La Coruña
{cgomezr, jvilares, alonso}@udc.es

Abstract

We present a technique for the construction of efficient prototypes for natural language parsing based on the compilation of parsing schemata to executable implementations of their corresponding algorithms. Taking a simple description of a schema as input, Java code for the corresponding parsing algorithm is generated, including schema-specific indexing code in order to attain efficiency.

Key words: parsing schemata, context-free grammars, tree-adjoining grammars

1 Introduction

The process of parsing, by which we obtain the structure of a sentence as a result of the application of grammatical rules, is a highly relevant step in the automatic analysis of natural language sentences. In the last decades, various parsing algorithms have been developed to accomplish this task. Although all of these algorithms essentially share the common goal of generating a tree structure describing the input sentence by means of a grammar, the approaches used to attain this result vary greatly between algorithms, so that different parsing algorithms are best suited to different situations.

Parsing schemata, described in [16], provide a formal, simple and uniform way to describe, analyze and compare different parsing algorithms. The notion of a parsing schema comes from considering parsing as a deduction process which generates intermediate results called *items*. An initial set of items is directly obtained from the input sentence, and the parsing process consists of the application of inference rules which produce new items from existing ones. Each item contains a piece of information about the sentence's structure, and a successful parsing process will produce at least one *final item* containing a full parse tree for the sentence or guaranteeing its existence.

In this paper, we will give a brief insight into the concept of parsing schemata by introducing a concrete example: a parsing schema for Earley's algorithm [5]. Given a context-free grammar $G = (N, \Sigma, P, S)$[1] and a sentence of length n which we denote by $a_1\ a_2\ \ldots\ a_n$, the schema describing Earley's algorithm is as follows[2]:

Item set:
$\{[A \to \alpha.\beta, i, j] \mid A \to \alpha\beta \in P \wedge 0 \le i < j\}$

Initial items (hypotheses):
$\{[a_i, i-1, i] \mid 0 < i \le n\}$

Deductive steps:

Earley Initter: $\dfrac{}{[S \to .\alpha, 0, 0]}\ S \to \alpha \in P$

Earley Scanner: $\dfrac{[A \to \alpha.a\beta, i, j] \qquad [a, j, j+1]}{[A \to \alpha a.\beta, i, j+1]}$

Earley Predictor: $\dfrac{[A \to \alpha.B\beta, i, j]}{[B \to .\gamma, j, j]}\ B \to \gamma \in P$

Earley Completer: $\dfrac{\begin{matrix}[A \to \alpha.B\beta, i, j] \\ [B \to \gamma., j, k]\end{matrix}}{[A \to \alpha B.\beta, i, k]}$

Final items:
$\{[S \to \gamma., 0, n]\}$

Items in the Earley algorithm are of the form $[A \to \alpha.\beta, i, j]$, where $A \to \alpha.\beta$ is a grammar rule with a special symbol (dot) added at some position in its right-hand side, and i, j are integer numbers denoting positions in the input string. The meaning of such an item can be interpreted as: "There exists a valid parse tree with root A, such that the direct children of A are the symbols in the string $\alpha\beta$, and the leaf nodes of the subtrees rooted at the symbols in α form the substring $a_{i+1} \ldots a_j$ of the input string".

The algorithm will produce a valid parse for the input sentence if an item of the form $[S \to \gamma., 0, n]$ is generated: according to the aforesaid interpretation, this item guarantees the existence of a parse tree with root S whose leaves are $a_1 \ldots a_n$, that is, 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

[1] Where N denotes the set of nonterminal symbols, Σ the set of terminal symbols, P the production rules and S the axiom.

[2] From now on, we will follow the usual conventions by which nonterminal symbols are represented by uppercase letters (A, $B \ldots$), terminals by lowercase letters (a, $b \ldots$) and strings of symbols (both terminals and nonterminals) by Greek letters (α, β...).

rules as in this example or specify other constraints that must be verified in order to infer the consequent.

2 Motivation

Parsing schemata are located at a higher abstraction level than algorithms. As can be seen in the example, a schema specifies the steps that must be executed and the intermediate results that must be obtained in order to parse a given string, but it makes no claim about the order in which to execute the steps or the data structures to use for storing the results.

Their abstraction of low-level details makes parsing schemata very useful, allowing us to define parsers in a simple and straightforward way. Comparing parsers, or considering aspects such as their correction and completeness or their computational complexity, also becomes easier if we think in terms of schemata. However, when we want to actually test a parser by running it on a computer and checking its results, we need to implement it in a programming language, so we have to abandon the high level of abstraction and worry about implementation details that were irrelevant at the schema level.

The technique presented in this paper automates this task, by compiling parsing schemata to Java language implementations of their corresponding parsers. The input to the compiler is a simple and declarative representation of a parsing schema, which is practically equal to the formal notation that we used previously. For example, a valid schema file describing the Earley parser is:

```
@goal [ S -> alpha . , 0 , length ]

@step EarleyInitter
------------------------ S -> alpha
[ S -> . alpha , 0 , 0 ]

@step EarleyScanner
[ A -> alpha . a beta , i , j ]
[ a , j , j+1 ]
---------------------------------
[ A -> alpha a . beta , i , j+1 ]

@step EarleyCompleter
[ A -> alpha . B beta , i , j ]
[ B -> gamma . , j , k ]
---------------------------------
[ A -> alpha B . beta , i , k ]

@step EarleyPredictor
[ A -> alpha . B beta , i , j ]
-------------------------- B -> gamma
[ B -> . gamma , j , j ]
```

3 Compiling Parsing Schemata

The compilation process, which transforms a declarative description of a parsing schema into a Java implementation of its corresponding parser, proceeds according to the following principles:

- A class is generated for each deductive step in the schema.
- The generated implementation will create an instance of this class for each possible set of values satisfying the side conditions that refer to production rules. For example, a distinct instance of the Earley PREDICTOR step will be created for each grammar rule of the form $B \rightarrow \gamma \in P$, which is specified in the step's side condition.
- The classes representing deductive steps have an `apply` method which tries to apply the deductive step to a given item. If the step is in fact applicable to the item (as determined by checking if the given item matches any of the step's antecedents), the method returns the new items obtained from the inference once all combinations of previously-generated items that satisfy the rest of the antecedents have been found.
- In order for our implementations to maintain the theoretical complexity of parsing algorithms, two distinct kind of indexes are generated for each schema: *existence indexes*, used to check whether an item exists in the item set, and *search indexes*, used to search for items conforming to a given specification. Apart from items, deductive steps are also indexed in *deductive step indexes*. These indexes are used to restrict the set of "applicable deductive steps" for a given item, discarding those known not to match it. Deductive step indexes usually have no influence on computational complexity with respect to input string size, but they do have an influence on complexity with respect to the size of the grammar, since the number of deductive step instances depends on grammar size when production rules are used as side conditions. All the generated indexing code is placed into two classes (the *item handler* and the *deductive step handler*) whose function is to provide efficient access to items and deductive steps, responding to queries issued by the deductive parsing engine. The indexing mechanism is explained in detail in [9].
- The execution of deductive steps in the generated code is coordinated by a *deductive parsing engine* [15] as described by the pseudocode in Figure 1. This is a schema-independent algorithm, and therefore its implementation is the same for any schema. It works with the set of all items that have been generated (either as initial hypotheses or as a result of the application of deductive steps) and an *agenda*, implemented as a queue, which contains the items we have not yet tried to trigger new deductions with. When the agenda is emptied, all possible items will have been generated, and the presence or absence of final items in the item set at this point indicates whether or not the input sentence belongs to the language defined by the grammar.

4 Parsing Context-Free Grammars

We have used our technique to generate implementations of three popular parsing algorithms for context-free grammars: CYK [11, 18], Earley and Left-Corner [12].

The schemata we have used describe recognizers, and therefore their generated implementation only

```
steps = {deductive step instances};
items = {initial items};
agenda = [initial items];
for each deductive step with an empty antecedent (s) in steps {
 result = s.apply([]);
 items.add(result);
 agenda.enqueue(result);
 steps.remove(s);
}
while agenda not empty {
 curItem = agenda.removeFirst();
 for each deductive step applicable to curItem (p) in steps {
  result = p.apply(curItem);
  items.add(result);
  agenda.enqueue(result);
 }
}
return items;
```

Fig. 1: *Pseudocode of the deductive parsing engine*

checks sentences for grammaticality by launching the deductive engine and testing for the presence of final items in the item set. However, these schemata can easily be modified to produce a parse forest as output [3]. If we want to use a probabilistic grammar in order to modify the schema so that it produces the most probable parse tree, this requires slight modifications of the deductive engine, since it should only choose the item with the highest probability when several items are available to match an antecedent.

The three algorithms have been tested with sentences from three different natural language grammars: the English grammar from the Susanne corpus [13], the Alvey grammar [4] (which is also an English-language grammar) and the Deltra grammar [14], which generates a fragment of Dutch. The Alvey and Deltra grammars were converted to plain context-free grammars by removing their arguments and feature structures. The test sentences were randomly generated by starting with the axiom and randomly selecting nonterminals and rules to perform expansions, until valid sentences consisting only of terminals were produced. Note that, as we are interested in measuring and comparing the performance of the parsers, not the coverage of the grammars; randomly-generated sentences are a good input in this case: by generating several sentences of a given length, parsing them and averaging the resulting runtimes, we get a good idea of the performance of the parsers for sentences of that length.

For Earley's algorithm, we have used the schema file described earlier. For the CYK algorithm, grammars were converted to Chomsky normal form (CNF), since this is a precondition of the algorithm. In the case of the Deltra grammar, which is the only one of our test grammars containing epsilon rules, we have used a weak variant of CNF allowing epsilon rules. For the Left-Corner parser, the schema used is the *sLC* variant described in [16].

The experiments are described in detail in [8]. The following conclusions can be drawn from them:

- The empirical computational complexity of the three algorithms is below their theoretical worst-case complexity of $O(n^3)$, where n denotes the length of the input string. In the case of the Susanne grammar, the measurements we obtain are close to being linear with respect to string size. In the other two grammars, the measurements grow faster with string size, but are still far below the cubic worst-case bound.

- CYK is the fastest algorithm in all cases, and it generates less items than the other ones. This may come as a surprise at first, as CYK is generally considered slower than Earley-type algorithms, particularly than Left-Corner. However, these considerations are based on time complexity relative to string size, and do not take into account complexity relative to grammar size. In this aspect, CYK is better than Earley-type algorithms, providing linear - $O(|P|)$ - worst-case complexity with respect to grammar size, while Earley is $O(|P|^2)$.[3] Therefore, the fact that CYK outperforms the other algorithms in our tests is not so surprising, as the grammars we have used have a large number of productions. The greatest difference between CYK and the other two algorithms in terms of the amount of items generated appears with the Susanne grammar, which has the largest number of productions. It is also worth noting that the relative difference in terms of items generated tends to decrease when string length increases, at least for Alvey and Deltra, suggesting that CYK could generate more items than the other algorithms for larger values of n.

[3] It is possible to reduce the computational complexity of Earley's parser to linear with respect to the grammar size by defining a new set of intermediate items and transforming accordingly prediction and completion deduction steps. Even in this case, CYK performs better that Earley's algorithm due to the lower number of items generated: $O(|N \cup \Sigma|\ n^2)$ for CYK vs. $O(|G|\ n^2)$ for Earley's algorithm, where $|G|$ denotes the size of the grammar measured as $|P|$ plus the summation of the lengths of all productions.

- Left-Corner is notably faster than Earley in all cases, except for some short sentences when using the Deltra grammar. The Left-Corner parser always generates fewer items than the Earley parser, since it avoids unnecessary predictions by using information about left-corner relationships. The Susanne grammar seems to be very well suited for Left-Corner parsing, since the number of items generated decreases by an order of magnitude with respect to Earley. On the other hand, the Deltra grammar's left-corner relationships seem to contribute less useful information than the others', since the difference between Left-Corner and Earley in terms of items generated is small when using this grammar. In some of the cases, Left-Corner's runtimes are a bit slower than Earley's because this small difference in items is not enough to compensate for the extra time required to process each item due to the extra steps in the schema, which make Left-Corner's matching and indexing code more complex than Earley's.

- The parsing of the sentences generated using the Alvey and Deltra grammars tends to require more time, and the generation of more items, than that of the Susanne sentences. This happens in spite of the fact that the Susanne grammar has more rules. The probable reason is that the Alvey and Deltra grammars have more ambiguity, since they are designed to be used with their arguments and feature structures, and information has been lost when these features were removed from them. On the other hand, the Susanne grammar is designed as a plain context-free grammar and therefore its symbols contain more information.

5 Parsing Tree-Adjoining Grammars

Although all the examples we have seen so far correspond to context-free parsing, our compilation technique is not limited to working with context-free grammars, since parsing schemata can be used to represent parsers for other grammar formalisms as well. All grammars in the Chomsky hierarchy can be handled in the same way as context-free grammars, and other formalisms can be added by defining element classes for their rules using the extensibility mechanism included in the system for defining new kinds of objects to use in schemata. The code generator can deal with these user-defined objects as long as some simple and well-defined guidelines are followed in their specification.

In particular, we have also used our system to generate parsers for tree-adjoining grammars [10]. A tree-adjoining grammar (TAG) includes a set of *elementary trees* of arbitrary depth which can be combined by using the *substitution* and *adjunction* operations. The substitution operation is used to substitute an elementary tree for a leaf node (which must be labelled as a *substitution node*) in another elementary tree. The adjunction operation allows us to insert an *auxilliary tree* (an elementary tree with a distinguished frontier node, called the *foot node* and labelled with the same nonterminal as its root) into another elementary tree.

The possibility of using elementary trees of arbitrary depth and the adjunction operation provide an extended domain of locality with respect to context-free grammars, and the set of languages which can be recognized with TAG is a strict superset of context-free languages. This makes TAG an interesting formalism for natural language parsing, since some phenomena present in natural languages cannot be represented by context-free grammars.

We have used our compiler to generate implementations for some of the most popular parsers for tree adjoining grammars [1, 2]: a CYK-based algorithm, two extensions of Earley's algorithm with and without the valid prefix property, and Nederhof's parsing algorithm. These implementations were tested both with artificially-generated grammars and a real-life, wide-coverage Feature-Based Tree Adjoining Grammar: the XTAG English grammar [17].

The TAG parsing schemata can be written in a format readable by our compiler in the same way as the context-free parsing schemata seen in the previous sections. Although the main constituents of TAG's are elementary trees instead of productions, each elementary tree may be expressed as a set of productions which can be used as side conditions for deductive steps. In order for the steps to be able to check whether the adjunction or substitution operation is allowed at a given node, we define boolean expressions that query the grammar for this information. In the case of the XTAG, we also need to include feature structures inside items and add unification operations to the deductive steps.

The performance results obtained from TAG parsers show that both string length and grammar size can be important factors in performance, and the interactions between them sometimes make their influence hard to quantify. The influence of string length in practical cases is usually below the theoretical worst-case bounds (we found the empirical complexity to be around $O(n^3)$, while the worst-case bound for these TAG parsers is $O(n^6)$). Grammar size becomes the dominating factor in large TAG's such as the XTAG, making tree filtering techniques advisable in order to achieve faster execution times.

By comparing performance of TAG and CFG parsers on artificially-generated grammars generating the same languages, we could see that using TAG's to parse context-free languages causes a significant overhead both in practical computational complexity and in constant factors, increasing execution times by several orders of magnitude with respect to CFG parsers.

A detailed explanation of the performance results obtained by applying our compilation technique to TAG parsers can be found at [6, 7].

6 Conclusions and future work

The construction of efficent prototypes directly from parsing schemata is very useful for the design, analysis and comparison of parsing algorithms, as it allows us to test them and check their results and performance without having to implement them in a programming language. As we have seen by comparing the performance of several well-known parsers for natural language grammars (context-free grammars and tree-adjoining grammars), not all algorithms are equally suitable for all grammars. In this work we provide a quick way to evaluate several parsing algorithms in order to find the best one for a particular application.

Currently, we are applying our compilation technique to automatically derive robust, error-correcting parsers from standard parsers for context-free grammars and tree adjoining grammars.

Acknowledgments

Supported in part by Ministerio de Educación y Ciencia (MEC) and FEDER (TIN2004-07246-C03-01, TIN2004-07246-C03-02), Xunta de Galicia (PGIDIT05PXIC30501PN, PGIDIT05PXIC10501PN, Rede Galega de Procesamento da Linguaxe e Recuperación de Información) and Programa de Becas FPU (MEC).

References

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

[2] M. A. Alonso, E. de la Clergerie, V. J. Díaz and M. Vilares. Relating tabular parsing algorithms for LIG and TAG. In H. Bunt, John Carroll and G. Satta (eds.), *New Developments in Parsing Technology*, pages 157-184, Kluwer Academic Publishers, Dordrecht-Boston-London, 2004.

[3] S. Billot and B. Lang. The structure of shared forest in ambiguous parsing. In *Proc. of ACL'89*, pages 143–151, Vancouver, British Columbia, Canada, 1989.

[4] J.A. Carroll. Practical unification-based parsing of natural language. Technical Report no. 314, University of Cambridge, Computer Laboratory, England. PhD Thesis., 1993.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[6] C. Gómez-Rodríguez, M.A. Alonso, and M. Vilares. On theoretical and practical complexity of TAG parsers. In P. Monachesi, G. Penn, G. Satta and S. Wintner (eds.), *FG 2006: The 11th conference on Formal Grammar. Malaga, Spain, July 29-30, 2006*, chapter 5, pp. 61-75, CSLI, Stanford, 2006.

[7] C. Gómez-Rodríguez, M.A. Alonso, and M. Vilares. Generating XTAG parsers from algebraic specifications. In *Proceedings of the 8th International Workshop on Tree Adjoining Grammar and Related Formalisms. Sydney, July 2006*, pp. 103-108, Association for Computational Linguistics, East Stroudsburg, PA, 2006.

[8] C. Gómez-Rodríguez, J. Vilares and M. A. Alonso. Compiling Declarative Specifications of Parsing Algorithms. In *Database and Expert Systems Applications*, volume of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin-Heidelberg-New York, 2007.

[9] C. Gómez-Rodríguez, M.A. Alonso, and M. Vilares. Generation of indexes for compiling efficient parsers from formal specifications. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia (eds.), *Computer Aided Systems Theory*, volume of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin-Heidelberg-New York, 2007.

[10] A.K. Joshi and Y, Schabes. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, eds, *Handbook of Formal Languages. Vol 3: Beyond Words*, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York, 1997.

[11] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab., Bedford, Massachussetts, 1965.

[12] D. J. Rosenkrantz and P. M. Lewis II. Deterministic Left Corner parsing. In *Conference Record of 1970 Eleventh Annual Meeting on Switching and Automata Theory*, pages 139–152, Santa Monica, CA, USA, 1970.

[13] G. Sampson. The Susanne corpus, Release 3, 1994.

[14] J. J. Schoorl and S. Belder. Computational linguistics at Delft: A status report, Report WT-M/TT 90–09, 1990.

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

[16] K. Sikkel. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms.* Springer-Verlag, Berlin/Heidelberg/New York, 1997.

[17] XTAG Research Group. A lexicalized tree adjoining grammar for English. Technical Report IRCS-01-03, IRCS, Univ. of Pennsylvania, 2001.

[18] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.