Compiling Declarative Specifications of Parsing Algorithms*

Carlos Gómez-Rodríguez, Jesús Vilares, and Miguel A. Alonso

Departamento de Computación, Universidade da Coruña (Spain)
{cgomezr, jvilares, alonso}@udc.es
Campus de Elviña, s/n - 15071 A Coruña (Spain)
Tel: +34 981 16 70 00 - Fax: +34 981 16 71 60

Abstract. The parsing schemata formalism allows us to describe parsing algorithms in a simple, declarative way by capturing their fundamental semantics while abstracting low-level detail. In this work, we present a compilation technique allowing the automatic transformation of parsing schemata to efficient executable implementations of their corresponding algorithms. Our technique is general enough to be able to handle all kinds of schemata for context-free grammars, tree adjoining grammars and other grammatical formalisms, providing an extensibility mechanism which allows the user to define custom notational elements.

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. Parsing schemata, described in [14], 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*. 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. An initial set of items is directly obtained from the input sentence, and the parsing process consists of the application of inference rules, called *deductive steps*, of the form $\frac{\eta_1 \dots \eta_m}{\xi}\ \Phi$ that allow us to infer the item specified by its consequent ξ from those in its antecedents $\eta_1 \dots \eta_m$. *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.

* Partially supported 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).

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 correctness 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 and check 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.

2 From declarative descriptions to program code

Our compilation process 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.
- 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, the method returns the new items obtained from the inference. In order to achieve this functionality, the method works as follows: first, it checks if the given item matches any of the step's antecedents. For every successful match found, the method searches for combinations of previously-generated items in order to satisfy the rest of the antecedents. Each combination of items satisfying all antecedents corresponds to an instantiation of the step variables which is used to generate an item from the consequent.
- The execution of deductive steps in the generated code is coordinated by a *deductive parsing engine*. This is a schema-independent algorithm, and therefore its implementation is the same for any schema:

```
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;
```

The algorithm 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. The correctness and completeness of this algorithm can easily be proved by induction. The parse forest can be recovered easily from the item set, as in [1].

2.1 Indexing

The implementation described above will only be efficient if we can efficiently access items and deductive steps. In particular, implementation of the operations checking if a given item exists in the item set (implicitly used by the `items.add` operation in the pseudocode above) and searching the item set for all items satisfying a certain specification (used by the `apply` method of deductive steps) affects the resulting parser's computational complexity. An inefficient implementation of any of these operations will give as result a parser with a computational complexity above the expected theoretical bounds for the corresponding algorithms. In order to maintain the theoretical complexity, we must provide constant-time access to items. In this case, each single deduction takes place in constant time, and the worst-case complexity is bounded by the maximum possible number of step executions: all complexity in the generated implementation is inherent to the schema.

In order to achieve this, we generate indexing code allowing efficient access to the item set. Two distinct kinds of indexes are generated for each schema, corresponding to the operations mentioned before: *existence indexes* are used to check whether an item exists in the item set, and *search indexes* allow us 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.

Our indexing mechanism is explained in detail in [7]. As an example of how the adequate indexes can be determined by a static analysis of the schema prior to compilation, we analyze the case where we have a deductive step of the form

$$\frac{[a, d, e, g] \qquad [b, d, f, g]}{(consequent)}\ c\ e\ f\ g$$

where each lowercase letter represents the set of elements (be them grammar symbols, string positions or other entities) appearing at particular positions in

the step, so that a stands for the set of elements appearing only in the first antecedent item, e represents those appearing in the first antecedent and side condition, g those appearing in both antecedents and side condition, and the rest of the letters represent the other possible combinations as can be seen in the step. In this example, we consider only two antecedents for the sake of simplicity, but the technique is general and can be applied to deductive steps with an arbitrary number of antecedents.

In this case, the following indexes are generated:

1. One deductive step index for each antecedent, using as keys the elements appearing both in the side condition *and* in that particular antecedent: therefore, two indexes are generated using the values (e, g) and (f, g). These indexes are used to restrict the set of deductive step instances applicable to items. As each instance corresponds to a particular instantiation of the side conditions, in this case each step instance will have different values for c, e, f and g. When the deductive engine asks for the set of steps applicable to a given item $[w, x, y, z]$, the deductive step handler will use the values of (y, z) as keys in order to return only instances with matching values of (e, g) or (f, g). Instances of the steps where these values do not match can be safely discarded, as we know that our item will not match any of both antecedents.
2. One search index for each antecedent, using as keys the elements appearing in that antecedent which are also present in the side condition *or* in the other antecedent. Therefore, a search index is generated by using (d, e, g) as keys in order to recover items of the form $[a, d, e, g]$ when d, e and g are known and a can take any value; and another index using the keys (d, f, g) is generated and used to recover items of the form $[b, d, f, g]$ when d, f and g are known. The first index allows us to efficiently search for items matching the first antecedent when we have already found a match for the second, while the second one can be used to search for items matching the second antecedent when we have started our deduction by matching the first one.
3. One existence index using as keys all the elements appearing in the consequent, since all of them are instantiated to concrete values when the step successfully generates a consequent item. This index is used to check whether the generated item already exists in the item set before adding it.

As this index generation process must be applied to all deductive steps in the schema, the number of indexes needed to guarantee constant-time access to items increases linearly with the number of steps. However, in practice we do not usually need to generate all of these indexes, since many of them are repeated or redundant. For example, if we suppose that the sets e and f in our last example contain the same number and type of elements, and elements are ordered in the same way in both antecedents, the two search indexes generated would in fact be the same, and our compiler would detect this fact and generate only one. In practical cases, the items used by different steps of a parsing schema usually have the same structure, so most indexes can be shared among several deductive steps and the amount of indexes generated is small.

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.

2.2 Elements in schemata

The variety of elements that may be present in parsing schemata poses an interesting difficulty if we want our technique to be general enough to cope with all sorts of schemata. The schemata notation is open, and any mathematical object could potentially appear as part of the definition of a schema.

As it is obviously impossible to provide a system that will recognize any kind of element that we could potentially include in a schema, but neither do we want our compiler to be limited to certain types of elements, we have defined an extensibility mechanism which allows us to define new elements that can be handled by the system in an easy way. For this purpose, we will classify all notational elements into four basic types, according to the treatment they should receive during code generation. Any new kind of element added to the system should be classified into one of these types:

- *Simple Elements:* Atomic, unstructured elements, which can be instantiated or not in a given moment. When simple elements are instantiated, they take a single value from a set of possible values, which can be bounded or not. Values can be converted to indexing keys. Examples of simple elements are grammar symbols, integers, string positions, probabilities...
- *Expression Elements*: These elements denote expressions which take simple elements or other expressions as arguments. For example, $i + 1$ is an expression element representing the sum of two string position arguments, and $tree[A, B]$ is an expression over nonterminal symbols. Feature structures and logic terms are also represented by this kind of elements. When all simple elements in an expression are instantiated to concrete values, the expression will be treated as a simple element whose value is obtained by applying the operation it defines (for example, summation). For the code generator to be able to do this, a Java expression must be provided as part of the expression element type definition, so that, for example, sums of string positions appearing in schemata can be converted to Java integer sums in the generated implementation. Unification of feature-structures has been implemented in this way.
- *Composite Elements*: Composite elements represent sequences of elements whose length must be finite and known. Composite elements are used to structure items. For instance, the Earley item $[A \rightarrow \alpha.B\beta, i, j]$ is represented as a composite element with three components: the first one is in turn a composite element, representing a grammar rule, while the remaining two are simple elements which denote string positions.
- *Sequence Elements*: These elements denote sequences of elements of any kind whose length is finite, but only becomes known when the sequence is

Table 1. Information about the grammars used in the experiments: total number of symbols, nonterminals, terminals, production rules, distribution of rule lengths, and average rule length.

Grammar	$\|N \cup \Sigma\|$	$\|N\|$	$\|\Sigma\|$	$\|P\|$	Epsilon	Unary	Binary	Other	Rule length
Susanne	1,921	1,524	397	17,633	0%	5.26%	22.98%	71.76%	3.54
Alvey	498	266	232	1,485	0%	10.64%	50.17%	39.19%	2.4
Deltra	310	282	28	704	15.48%	41.05%	18.18%	25.28%	1.74

instantiated to a concrete value. The strings appearing in Earley items are examples of sequence elements, being able to represent symbol strings of any length. The code generator must take this fact into account when generating matching code for these elements.

In order to add a new kind of element to the schema compiler, the user will have to define it as a subclass of one of these four basic types, and implement that type's interface by following some simple guidelines. In addition to this, the user must provide one or more regular expressions in order to specify the format of the strings representing the new kind of element in schemata definition files. These expressions can be included in a global configuration file or directly in the schema files that will use the element. The schema parser will use the regular expressions to identify our new type of element in schema files. When one of these elements is found in a schema, the compiler will dynamically load the corresponding class and instantiate it by using Java's reflection mechanisms, thus avoiding the need to recompile the system in order to add new element classes. This makes our technique highly extensible, and easily allows us to work with schemata containing all kinds of non-predefined items.

3 Experimental results

We have used our technique to generate implementations of three popular parsing algorithms for context-free grammars: CYK [9, 15], Earley [3] and Left-Corner [10]. The schemata we have used describe recognizers, and therefore their generated implementation only 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 [1]. 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 [11], the Alvey grammar [2] (which is also an English-language grammar) and the Deltra grammar [12], 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. Table 1 summarizes some facts about the three grammars, where by "Rule Length" we mean the average length of the right-hand side of a grammar's rules.

For Earley's algorithm, we have used the schema described in [14]. 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 [14].

Performance results[1] for all these algorithms and grammars are shown in table 2. The following conclusions can be drawn from the measurements:

- 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 $O(|P|^2)$. 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[2]. 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

[1] The machine used for these tests was a standard laptop: Intel 1500 MHz Pentium M processor, 512 MB RAM, Sun Java Hotspot virtual machine (version 1.4.2_01-b06) and Windows XP.

[2] It is possible to reduce the computational complexity of Earley's parser by applying some transformations to the schema. Even in this case, CYK performs better than Earley's algorithm due to the smaller number of items generated: $O(|N \cup \Sigma|n^2)$ for CYK vs. $O(|G|n^2)$ for Earley, where $|G|$ denotes the size of the grammar measured as the number of productions plus the summation of the lengths of all productions.

Table 2. Performance measurements for generated parsers.

Grammar	String length	Time Elapsed (s)			Items Generated		
		CYK	Earley	LC	CYK	Earley	LC
Susanne	2	0.000	1.450	0.030	28	14,670	330
	4	0.004	1.488	0.060	59	20,945	617
	8	0.018	4.127	0.453	341	51,536	2,962
	16	0.050	13.162	0.615	1,439	137,128	7,641
	32	0.072	17.913	0.927	1,938	217,467	9,628
	64	0.172	35.026	2.304	4,513	394,862	23,393
	128	0.557	95.397	4.679	17,164	892,941	52,803
Alvey	2	0.000	0.042	0.002	61	1,660	273
	4	0.002	0.112	0.016	251	3,063	455
	8	0.010	0.363	0.052	915	7,983	1,636
	16	0.098	1.502	0.420	4,766	18,639	6,233
	32	0.789	9.690	3.998	33,335	66,716	39,099
	64	5.025	44.174	21.773	133,884	233,766	170,588
	128	28.533	146.562	75.819	531,536	596,108	495,966
Deltra	2	0.000	0.084	0.158	1,290	1,847	1,161
	4	0.012	0.208	0.359	2,783	3,957	2,566
	8	0.052	0.583	0.839	6,645	9,137	6,072
	16	0.204	2.498	2.572	20,791	28,369	22,354
	32	0.718	6.834	6.095	57,689	68,890	55,658
	64	2.838	31.958	29.853	207,745	282,393	261,649
	128	14.532	157.172	143.730	878,964	1,154,710	1,110,629

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.

- 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.
- Execution times for the Alvey grammar quickly grow for sentence lengths above 16. This is because sentences generated for these lengths tend to be repetitions of a single terminal symbol, and are highly ambiguous.

4 Conclusions

In this paper, we have presented a compilation technique which allows us to automatically transform a parsing schema into an implementation of the algorithm it describes, keeping the theoretical computational complexity of the algorithm. This makes our work different from the parsing machine described by Shieber *et al.* in [13], a Prolog implementation of a deductive parsing engine which can also be used to implement parsing schemata; however, its input notation is less declarative, since schemata have to be programmed in Prolog, and it does not support automatic indexing, so the resulting parsers are inefficient unless the user programs indexing code by hand, abandoning the high abstraction level. Another alternative for implementing parsing schemata is the Dyna language [4], which can be used to implement some kinds of dynamic programs; but it has a complex notation, clearly less declarative than ours, which is specifically designed for denoting schemata: in our approach, the user only has to write the schema without worrying about implementation details. In addition, we provide an extensibility mechanism that allows the user to add new kinds of elements to schemata apart from the predefined ones.

Compilation of parsing schemata has been shown very useful for the design, analysis and prototyping of parsing algorithms, as it has allowed us to test them (even variants with "tricks" that improve practical performance in some cases) and check their results and performance without having to implement them in a programming language. As we have seen by comparing the performance of CYK, Earley and Left-Corner parsers for several 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.

Our compilation technique is not limited to working with context-free grammars, since 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. In this way, we have used our compiler to generate implementations for some of the most popular parsers for tree adjoining grammars (TAG) [8]. A detailed explanation of the performance results obtained by applying our compilation technique to TAG parsers can be found at [5, 6].

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

References

1. S. Billot and B. Lang. The structure of shared forest in ambiguous parsing. In *Proc. of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, Vancouver, British Columbia, Canada, June 1989. ACL.
2. J.A. Carroll. Practical unification-based parsing of natural language. Technical Report no. 314, University of Cambridge, Computer Laboratory, England. PhD Thesis, 1993.
3. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
4. J. Eisner, E. Goldlust, and N. A. Smith. Dyna: A declarative language for implementing dynamic programs. In *Proceedings of ACL 2004 (Companion Volume)*, pages 218–221, Barcelona, July 2004.
5. 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, Center for the Study of Language and Information, Stanford, 2006.
6. 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.
7. 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.
8. A. K. Joshi and Y. Schabes. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, chapter 2, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York, 1997.
9. 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.
10. 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, October 1970. IEEE.
11. G. Sampson. The Susanne corpus, Release 3, 1994.
12. J. J. Schoorl and S. Belder. Computational linguistics at Delft: A status report, Report WTM/TT 90–09, 1990.
13. 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.
14. K. Sikkel. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Springer-Verlag, Berlin/Heidelberg/New York, 1997.
15. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.