Generation of indexes for compiling efficient parsers from formal specifications

Carlos Gómez-Rodríguez[1], Miguel A. Alonso[1], and Manuel Vilares[2]

[1] Departamento de Computación, Universidade da Coruña
Facultad de Informtica, Campus de Elviña 5, 15071 La Coruña (Spain)
{cgomezr, alonso}@udc.es
[2] Departamento de Informática, Universidade de Vigo
E.T.S. de Ingeniería Informática, Campus As Lagoas, 32004 Orense (Spain)
vilares@uvigo.es

http://www.grupocole.org/

Abstract. Parsing schemata 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 (called *deductive steps*) 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. Their abstraction of low-level details makes parsing schemata useful 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 use a parser by running it on a computer, 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. In particular, we study in this article how the source parsing schema should be analysed to decide what kind of indexes need to be generated in order to obtain an efficient parser.

1 Introduction

Parsing schemata are explained in detail in [14, 11]. In this article, we will give a brief insight into the concept by introducing a concrete example: a parsing schema for Earley's algorithm [3]. Given a context-free grammar $G = (N, \Sigma, P, S)$, where N denotes the set of nonterminal symbols, Σ the set of terminal symbols, P the production rules and S the axiom of the grammar, and a sentence of length n denoted by $a_1\ a_2\ \ldots\ a_n$, the schema describing Earley's algorithm is as follows:[1]

[1] 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,

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

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

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: $\dfrac{[A \to \alpha.B\beta, i, j] \qquad [B \to \gamma., j, k]}{[A \to \alpha B.\beta, i, k]}$

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

1.1 Compilation of parsing schemata

Parsing schemata are located at a higher abstraction level than algorithms. As can be seen in the text above, a schema specifies the steps that must be executed

$b \ldots$) and strings of symbols (both terminals and nonterminals) by Greek letters (α, $\beta \ldots$).

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. To overcome this, we have developed a compiler that given a parsing schema is able to obtain an executable implementation of the corresponding parser [7, 6]. 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 would be:

```
@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 ]
```

1.2 Related work

Shieber *et al.* provide in [12] 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, the Dyna language [4], can be used to implement some kinds of dynamic programs; but it has a more complex and less declarative notation than ours, which is specifically designed for denoting schemata.

2 Automatic generation of indexes for parsing schemata

In this section, we study how the compiler analyses the source parsing schema to decide what kind of indexes need to be generated in order to obtain an efficient parser. In particular, implementation of the following operations affects the resulting parser's computational complexity:

- Check if a given item exists in the item set.
- Search the item set for all items satisfying a certain specification: Once an antecedent of a deductive step has been matched, a search for items matching the rest of the antecedents is needed in order to make inferences using that step.

In order to maintain the theoretical complexity of parsing schemata, 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 propose to generate two distinct kind of indexes for each schema, corresponding to the operations mentioned before:

- Items are indexed in *existence indexes*, used to check whether an item exists in the item set, and *search indexes*, which allow us to search for items conforming to a given specification.
- Deductive steps are 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 step instances depends on grammar size when grammar productions are used as side conditions.

2.1 A simple case

The generation of indexing code is not trivial, since the elements by which we should index items in order to achieve efficiency vary among schemata. For instance, if we are trying an Earley COMPLETER step

$$\frac{[A \to \alpha.B\beta, i, j], [B \to \gamma., j, k]}{[A \to \alpha B.\beta, i, k]}$$

on an item of the form $[B \to \gamma., j, k]$ which matches the second antecedent, we will need to search for items of the form $[A \to \alpha.B\beta, i, j]$, for any values of A, α, β and i, in order to draw all the possible conclusions from the item and step. Since the values of B and j are fixed, this search will be efficient and provide constant-time access to items if we have them indexed by the symbol that follows the dot and by the second string position (B and j). However, if we analyze the other Earley steps in the same way, we will find that their indexing needs are different, and different parsing schemata will obviously have different needs.

Therefore, in order to generate indexing code, we must take the distinct features of each schema into account. In the case of search indexes, we must analyze each deductive step just as we have analyzed the COMPLETER step: we must keep track of which variables are instantiated to a concrete value when a search must be performed. This information is known at schema compilation time and allows us to create indexes by the elements corresponding to instantiated variables. For example, in the case of COMPLETER, we would create the index that we mentioned before (by the symbol directly after the dot and the second string position) and another index by the symbol in the left side of productions and the first string position. This second index is useful when we have an item matching the first antecedent and we want to search for items matching the second one, and is obtained by checking which variables are instantiated when the first antecedent is matched.

The generation of existence indexes is similar to, but simpler than, that of search indexes. The same principle of checking which variables will be instantiated when the index is needed is valid in this case, but when an item is checked for existence it is always fully known, so all its variables are instantiated.

Deductive step indexes are generated by taking into account those step variables which will take a value during instantiation, i.e. which variables appear on side conditions. Since these variables will have a concrete value for each instance of the step, they can be used to filter instances in which they take a value that will not allow matching with a given item.

2.2 A general case

For a more general view of how the adequate indexes can be determined by a static analysis of the schema prior to compilation, we can analyze the general case where we have a deductive step of the form

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

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. We can easily see that any deductive step with two antecedent items can be represented in this way (note that the element sets can be empty, and that the ordering of elements inside items is irrelevant to this discussion).

In this case, the following indexes are generated:

- 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.
- 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.
- 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.

In our example, we have considered 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.

3 Experimental results

We have used our technique to generate implementations of three popular parsing algorithms for context-free grammars: CYK [8, 15], Earley and Left-Corner [9]. The three algorithms have been tested with sentences from three different natural language grammars: the English grammar from the Susanne corpus [10], the Alvey grammar [2] (which is also an English-language grammar) and the Deltra grammar [13], 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.[2] Performance results for all these algorithms and grammars are shown in Table 1.[3] The success of the index generation technique proposed in this article is shown by the fact that the empirical computational complexity of the three

[2] 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.

[3] Tests performed on a laptop with an Intel 1500 MHz Pentium M processor, 512 MB RAM, Sun Java Hotspot virtual machine (version 1.4.2_01-b06) and Windows XP.

Table 1. 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

parsers is below their theoretical worst-case complexity of $O(n^3)$. Similar results have been obtained for highly ambiguous artificial grammars such as the one used in [1].

4 Conclusions

Parsing algorithms can be defined in a simple, formal and uniform way by means of Parsing Schemata, and there exists a compilation technique which allows us to automatically transform a parsing schema into an implementation of the algorithm it describes. In this article we have shown how adapted indexing code can be automatically generated for parsing schemata, so that the generated parsers keep the theoretical computational complexity of the parsing algorithms.

Although all the cases we have seen so far correspond to context-free parsing, our 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. For example, we have generated implementations for some of the most popular parsers for Tree Adjoining Grammars [5, 6].

Acknowledgements

The work reported in this article has been 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 of MEC.

References

1. M. A. Alonso, D. Cabrero and M. Vilares. Construction of Efficient Generalized LR Parsers. *Lecture Notes in Computer Science*, 1436:7–24, 1998.
2. J.A. Carroll. Practical unification-based parsing of natural language. TR 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, J. Vilares, and M.A. Alonso. Automatic Generation of Natural Language Parsers from Declarative Specifications. In L. Penserini, P. Peppas and A. Perini (eds.), *STAIRS 2006 - Proceedings of the Third Starting AI Researchers' Symposium, Riva del Garda, Italy, August 2006*, vol. 142 of *Frontiers in Artificial Intelligence and Applications*, pp. 259-260, IOS Press, Amsterdam, 2006.
8. 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.
9. 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.
10. G. Sampson. The Susanne corpus, Release 3, 1994.
11. Karl-Michael Schneider. *Algebraic Construction of Parsing Schemata*. Mensch & Buch Verlag, Berlin, Germany, 2000.
12. 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.
13. J. J. Schoorl and S. Belder. Computational linguistics at Delft: A status report, Report WTM/TT 90–09, 1990.
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.