Automatic Generation of Natural Language Parsers from Declarative Specifications[1]

Carlos GÓMEZ-RODRÍGUEZ [a], Jesús VILARES [b] and Miguel A. ALONSO [b]
[a] *Escuela Superior de Ingeniería Informática, Universidade de Vigo (Spain)*
e-mail: cgomezr@uvigo.es
[b] *Departamento de Computación, Universidade da Coruña (Spain)*
e-mail: {jvilares, alonso}@udc.es

Abstract. The parsing schemata formalism allows us to describe parsing algorithms in a simple 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 executable implementations of their corresponding algorithms. Taking a simple description of a schema as input, our technique generates Java code for the corresponding parsing algorithm, including schema-specific indexing code in order to attain efficiency. Our technique is general enough to be able to handle all kinds of schemata for context-free 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. 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 [16,13], 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

[1]Partially supported by Ministerio de Educación y Ciencia and FEDER (Grants TIN2004-07246-C03-01, TIN2004-07246-C03-02), and Xunta de Galicia (Grants PGIDIT05PXIC30501PN, PGIDIT05PXIC10501PN and PGIDIT05SIN044E).

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.

Almost all known parsing algorithms may be described by a parsing schema (non-constructive parsers, such as those based on neural networks, are exceptions). This is done by identifying the kinds of items that are used by a given algorithm, defining a set of inference rules describing the legal ways of obtaining new items, and specifying the set of final items whose presence indicates that the sentence has been parsed.

1.1. Parsing Schemeta

Sikkel [16] provides an accurate and detailed explanation of parsing schemata. In this paper, we will give a brief insight into the concept by introducing a concrete example: a parsing schema for Earley's algorithm [4]. Given a context-free grammar $G = (N, \Sigma, P, S)$[2] 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[3]:

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

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

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

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 $[S \rightarrow \gamma., 0, n]$ 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.

1.2. 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 will 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 ]
```

1.3. Related work

Shieber *et al.* provide in [15] 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 [5], 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. From declarative descriptions to program code

Our 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, 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*, as described in [15]. This is a schema-independent algorithm, and therefore its implementation is the same for any schema, as described by the following pseudocode:

```
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*, im-

plemented 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 correction and completeness of this algorithm can easily be proved by induction. The parse forest can be recovered easily from the item set, as in [2].

2.1. Indexing

The combination of the deductive parsing engine with the code associated to each deductive step provides a full implementation of the parser described by the schema. However, this implementation will only be efficient if we can efficiently access items and deductive steps. In particular, implementation of the following operations affects the resulting parser's computational complexity:

- Check if a given item exists in the item set. This operation is implicitly used by the `items.add(...)` operation in the pseudocode above.
- Search the item set for all items satisfying a certain specification. This operation is used by the `apply` method of deductive steps: once an antecedent has been matched, a search for items matching the rest of the antecedents is needed in order to make inferences using the step.

If we provided an inefficient implementation of any of these operations (such as searching for items by sequentially traversing the set and individually checking if each item conforms to the specification), our generated implementations would have a computational complexity above the expected theoretical bounds for the corresponding algorithms. This fact can be easily checked by studying particular algorithms, such as Earley or CYK [7,19]. 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 kind 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.

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 to execute an Earley COMPLETER step on an item of the form $[B \rightarrow \gamma., j, k]$, which matches the second antecedent, we will need to search for items of the form $[A \rightarrow \alpha.B\beta, i, j]$, 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. 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 kinds of elements which may appear in a schema are not limited to the ones we have seen in the Earley example: 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, the dot in Earley items...
- *Expression Elements*: These elements denote expressions which take simple elements or other expressions as arguments. For example, $i + 1$ is an expression ele-

ment 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 instantiated to a concrete value. The strings α, β and γ appearing in the Earley schema 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

In this section, we will present some examples of the use of our compilation technique with different algorithms and grammars, and their corresponding performance measurements. In particular, we have used our technique to generate implementations of three popular parsing algorithms for context-free grammars: CYK, Earley and Left-Corner [10]. However, we must remark that we are 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

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

classes for their rules using the extensibility mechanism. For example, we have also used the system to generate parsers for tree adjoining grammars [6].

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 [2]. 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 [3] (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. 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 file described earlier[4].

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 and the following CYK variant, which can handle them:

[4] The results in fact correspond to a slightly modified Earley schema, which "divides" the PREDICTOR step into different steps for different lengths of the associated production's right side (0, 1, 2 or more symbols). This modification of the schema makes existence indexing more effective and thus reduces execution times, without affecting computational complexity.

```
@goal [ S , 0 , length ]

@step CYKBinary
[ B , i , j ]
[ C , j , k ]
--------------------- A -> B C
[ A , i , k ]
```

```
@step CYKUnary
[ a , i , j ]
--------------------- A -> a
[ A , i , j ]

@step CYKEpsilon
[ epsilon , i , i ]
--------------------- A ->
[ A , i , i ]
```

For the Left-Corner parser, the schema used is the sLC variant described in [16], where side conditions containing boolean expression elements[5] are used to evaluate left-corner relationships:

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

@step SimplifiedLCInitter
---------------------------- S -> gamma
[ S -> . gamma , 0 , 0 ]

@step SimplifiedLCRuleToItem
---------------------------- B -> beta
[ B -> beta ]

@step SimplifiedLCNonterminal
[ E , i ]
[ A -> alpha . , i , j ]
[ B -> A beta ]
------------------------ / LC(E;B)
[ B -> A . beta , i , j ]

@step SimplifiedLCTerminal
[ E , i ]
[ a , i , i+1 ]
[ B -> a beta ]
------------------------- / LC(E;B)
[ B -> a . beta , i , i+1 ]
```

```
@step SimplifiedLCEpsilon
[ E , i ]
[ B -> ]
------------------------------
[ B -> . , i , i ]

@step SimplifiedLCPredictor
[ C -> gamma . E delta , k , i ]
------------------------------
[ E , i ]

@step SimplifiedLCScanner
[ A -> alpha . B beta , i , j ]
[ B , j , j+1 ]
------------------------------
[ A -> alpha B . beta , i , j+1 ]

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

Performance results[6] 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 con-

[5] Side conditions of this kind are compiled to code that prevents the steps from generating results if the corresponding predicates don't hold. This code is inserted in such a way that the predicates are checked as soon as the referenced variables are instantiated. By aborting the steps as soon as possible when predicates don't hold, we avoid unnecessary calculations.

[6] The machine used for these tests was a standard end-user computer, a laptop equipped 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 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

siderations are based on time complexity relative to string size, and don't 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. 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.

- 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 and future work

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. We have seen examples of the application of this technique to several schemata associated to context-free grammars (CYK, Earley and Left-Corner) but it can also be applied to other grammar formalisms.

Our input notation is highly declarative, so the user only has to write the schema without worrying about implementation details. An extensibility mechanism allows the user to add new kinds of elements to schemata apart from the predefined ones. Adapted indexing code is automatically generated for each schema, so that generated implementations keep the theoretical computational complexity of the algorithms.

Compilation of parsing schemata is very useful for the design, analysis and prototyping 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 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.

Currently, we are applying our compilation technique in two directions:

- To analyze parsing performance for grammars based on other formalisms used for natural language processing, such as the XTAG [18], a wide coverage Feature-Based Tree-Adjoining Grammar (FB-TAG) [6]. In this context, we are studying the behaviour of some of the most popular parsers for tree adjoining grammars: a CYK-based algorithm [17], two extensions of the Earley's algorithm with and without the valid prefix property [12,1], Nederhof's parsing algorithm [8] and Van Noord's head-corner parser [9].

- To generate robust parsers for context-free grammars and tree adjoining grammars.

References

[1] Miguel A. Alonso, David Cabrero, Eric de la Clergerie, and Manuel Vilares. Tabular algorithms for TAG parsing. In *Proc. of EACL'99, 9th Conference of the European Chapter of the Association for Computational Linguistics*, pages 150–157, Bergen, Norway, June 1999. ACL.

[2] Sylvie Billot and Bernard 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.

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

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

[5] Jason Eisner, Eric Goldlust, and Noah A. Smith. Dyna: A declarative language for implementing dynamic programs. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (Companion Volume)*, pages 218–221, Barcelona, July 2004.

[6] Aravind K. Joshi and Yves 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.

[7] 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.

[8] Mark-Jan Nederhof. The computational complexity of the correct-prefix property for TAGs. *Computational Linguistics*, 25(3):345–360, 1999.

[9] Gertjan van Noord. Head-corner parsing for TAG. *Computational Intelligence*, 10(4):525–534, 1994.

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

[13] Karl-Michael Schneider. *Algebraic Construction of Parsing Schemata*. Mensch & Buch Verlag, Berlin, Germany, 2000.

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

[15] Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36, July-August 1995.

[16] Klaas Sikkel. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science — An EATCS Series. Springer-Verlag, Berlin/Heidelberg/New York, 1997.

[17] K. Vijay-Shanker and Aravind K. Joshi. Some computational properties of tree adjoining grammars. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 82–93, Chicago, IL, USA, July 1985. ACL.

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

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