SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–6 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

A compiler for parsing schemata

C. Gómez-Rodríguez[1]*, J. Vilares[1]*, M. A. Alonso[1]*†

[1] *Depto. de Computación, Facultade de Informática, Universidade da Coruña. Campus de Elviña, s/n, 15071 A Coruña, SPAIN.*

SUMMARY

We present a compiler which can be used to automatically obtain efficient Java implementations of parsing algorithms from formal specifications expressed as parsing schemata. The system performs an analysis of the inference rules in the input schemata in order to determine the best data structures and indexes to use, and ensure that the generated implementations are efficient. The system described is general enough to be able to handle all kinds of schemata for different grammar formalisms, such as context-free grammars and tree-adjoining grammars, and it provides an extensibility mechanism allowing the user to define custom notational elements. This compiler has proven very useful for analyzing, prototyping and comparing natural language parsers in real domains, as can be seen in the empirical examples provided at the end of the article. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: Parsing; declarative programming; natural language processing; parsing schemata.

NOTICE: this is a preprint of an article accepted for publication in *Software: Practice & Experience*, Copyright © 2008 John Wiley & Sons, Ltd. Changes resulting from the publishing process, such as editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. A definitive version was subsequently published in Software: Practice & Experience, DOI: 10.1002/spe.904, available at http://www3.interscience.wiley.com/journal/1752/home

*Correspondence to: Depto. de Computación, Facultade de Informática, Universidade da Coruña. Campus de Elviña, s/n, 15071 A Coruña, SPAIN. Email: {cgomezr | jvilares | alonso}@udc.es

Contract/grant sponsor: MEC and FEDER; contract/grant number: TIN2004-07246-C03, HUM2007-66607-C04
Contract/grant sponsor: Xunta de Galicia; contract/grant number: PGIDIT07SIN005206PR, PGIDIT05PXIC-10501PN, PGIDIT05PXIC30501PN
Contract/grant sponsor: Rede Galega de Procesamento da Linguaxe e Recuperación de Información

1. INTRODUCTION

The process of *parsing*, or analyzing a sequence of tokens to obtain its internal structure, is a highly relevant step in many software systems. Compilers need to parse source code, written in a programming language, in order to convert it into instructions executable by a machine. Systems that perform natural language processing tasks (such as machine translation, information extraction or text summarization) apply parsing to sentences in human language.

Natural language parsers have to cope with *ambiguity*: many sentences in human languages have more than one viable interpretation, corresponding to more than one parse tree. On the other hand, grammars describing programming languages are generally designed to be *unambiguous*, so that every program has at most one valid interpretation. Programming language parsers usually take advantage of this fact and are more efficient, although less general, than those used for natural language.

In recent decades, various parsing algorithms have been developed. Although all of them share the common goal of generating a hierarchical description of the input (often by means of a *grammar*, a set of rules describing the language), the approaches used to attain this result vary greatly between algorithms, so that different parsing algorithms are best suited to different situations. In particular, if we focus only on natural language parsers we find a wide variety of algorithms, and the choice of the best one for a particular application will depend heavily on the characteristics of the grammar and sentences it is to work with [28, 7].

The parsing schemata formalism, introduced by Sikkel [28], provides a formal way to capture the essential features of a parser while abstracting implementation details. The parsing schemata framework is based on the idea of viewing parsing as a deduction process. Parsing schemata are highly declarative descriptions of parsers, as they specify *what* to do (a set of operations to be performed on intermediate results) but not *how* to do it (the order in which to execute the operations, or the data structures used to store the results).

As an example, one of the most widely used parsing algorithms for natural language processing is Earley's algorithm [10]. The original paper defines the parser in an algorithmic fashion, as in the pseudocode in figure 1. A parsing schema for this parser, appearing in [28], is shown in figure 2. Regardless of the concrete semantics of the schema (which will be addressed later), it is obvious at a glance that the schema provides a much simpler and more straightforward description than the algorithm, even in the form of high-level pseudocode.

A parsing schema can be seen as a formal *specification* of a parser's behavior, which can be implemented in several ways. Almost all known parsing algorithms may be described by a parsing schema (nonconstructive parsers, such as those based on neural networks, are exceptions). This generality, along with their simplicity and high abstraction level, makes parsing schemata a useful tool for defining, analyzing and comparing parsers. 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 system presented in this article automates this task, by compiling parsing schemata to efficient Java language implementations of the corresponding algorithms. The input to the compiler is a declarative specification of a parser in the form of a parsing schema, and the output is an efficient implementation of the parser. This enables us to save a lot of work, since we can test parsers and check their results and performance just by writing their specification, without having to implement them.

```
S = **array** [0.. $n$] of state sets ;
**for** $i$ = 0 ... $n$ { S[$i$] = $\emptyset$; } //initialize the $n$+1 sets to $\emptyset$
**for** each rule $S \to \alpha \in P$ //initialize S[0]
  S[0] = S[0] $\cup \{(S \to \alpha, 0, 0)\}$;

**for** $i$ = 0 ... $n$ { //process state sets
  process the members of S[$i$] in order, executing each of these operations on
  each state $(A \to \alpha, j, f)$ until no more of them can be applied:
    1) Predictor :
      $X$ = $j+1$th symbol in $\alpha$;
      **if** $X$ exists and is a nonterminal
        **for** each production of the form $X \to \beta$ in $P$
          S[$i$] = S[$i$] $\cup \{(X \to \beta, 0, i)\}$;
    2) Completer:
      **if** $X$ does not exist //($j+1 > |\alpha|$)
        **for** each state $(B \to \beta, l, g)$ in S[$f$] {
          $Y$ = $l+1$th symbol in $\beta$;
          **if** $Y$ exists $\wedge\ Y = A$
            S[$i$] = S[$i$] $\cup \{(B \to \beta, l+1, g)\}$;
        }
    3) Scanner:
      **if** $X$ exists and is a terminal
        **if** $X$ = $a_{i+1}$
          S[$i+1$] = S[$i+1$] $\cup \{(A \to \alpha, j+1, f)\}$;
}

// check whether string belongs to language
**if** S[$n$] contains a state of the form $(S \to \gamma, |\gamma|, 0)$ **return true**;
else **return false** ;
```

Figure 1. Pseudocode for Earley's parsing algorithm [10] for a string of length n. The algorithm works on states of the form $(A \to \alpha, j, k)$, where $A \to \alpha$ is the grammar rule currently being used for recognition, j is the number of already recognized symbols in its right-hand side, and k is the initial position of the part of the input string which has been recognized (the final position is given by the index of the state set S[i] holding the state).

This can be useful both for designing new algorithms and for testing existing ones to determine which is the best for a particular application. The source code and binaries of the system can be downloaded from `http://www.grupocole.org/software/COMPAS/`.

In the next section, we provide a brief introduction to the parsing schemata formalism, limited only to the core concepts. This allows us to explain our system's goal in more detail, and discuss related work directed towards similar goals. We then proceed to explain how the system is implemented, with sections discussing the architecture of the generated code, the reading of schemata and the generation process, including the generation of indexes. Finally, we show some examples of its use including performance measurements with well-known natural language grammars.

Item set: $\{[A \rightarrow \alpha.\beta, i, j] \mid A \rightarrow \alpha\beta \in P \wedge 0 \leq i < j\}$
Initial items (hypotheses): $\{[a_i, i-1, i] \mid 0 < i \leq n\}$

Deduction steps:

INITTER: $\dfrac{}{[S \rightarrow .\alpha, 0, 0]}\; S \rightarrow \alpha \in P$

SCANNER: $\dfrac{[A \rightarrow \alpha.a\beta, i, j] \qquad [a, j, j+1]}{[A \rightarrow \alpha a.\beta, i, j+1]}$

PREDICTOR: $\dfrac{[A \rightarrow \alpha.B\beta, i, j]}{[B \rightarrow .\gamma, j, j]}\; B \rightarrow \gamma \in P$

COMPLETER: $\dfrac{\begin{array}{c}[A \rightarrow \alpha.B\beta, i, j]\\ [B \rightarrow \gamma., j, k]\end{array}}{[A \rightarrow \alpha B.\beta, i, k]}$

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

Figure 2. A parsing schema specifying Earley's parsing algorithm.

2. PARSING SCHEMATA

2.1. Languages and grammars

A *language* is a set of sequences (strings) of symbols from a finite set called an *alphabet*.

A *grammar* is a precise definition of a language by means of a set of rules. One of the most widely used types of grammar is context-free grammars. A *context-free grammar* is a 4-tuple $G = (N, \Sigma, P, S)$ where:

- Σ is an alphabet of symbols called *terminal symbols*, which will be the components of the strings in the language associated with G,
- N is an alphabet of auxiliary symbols called *nonterminal symbols*, which will not appear in the strings of the language,
- $S \in N$ is a special nonterminal symbol called the *initial symbol* or *axiom* of G,
- $P \subseteq N \times (\Sigma \cup N)^*$ is a set of *production rules* of the form $A \rightarrow \alpha$, where A is a nonterminal symbol and α is a string that may contain both terminal and nonterminal symbols.

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 $(\alpha, \beta \ldots)$.

The language associated with a grammar $G = (N, \Sigma, P, S)$, denoted $L(G)$, is the set of strings of terminal symbols in Σ that can be obtained by starting with the initial symbol S and applying a sequence of productions in P. A production of the form $A \rightarrow \alpha$ can be applied to any string containing the nonterminal A, and is applied by changing one appearance of A in the string to α. We write $\beta \Rightarrow \gamma$ to denote that we can obtain the string γ by applying a production to the string β.

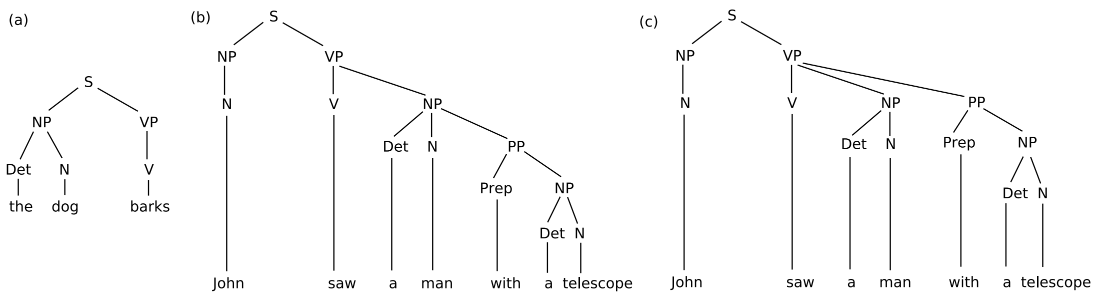

Figure 3. (a) Parse tree for a simple sentence. (b) and (c): Two alternative parse trees for an ambiguous sentence.

Example: Suppose that we have a context-free grammar $G = (N, \Sigma, P, S)$ defined by the following:

- $\Sigma = \{$the,dog,barks$\}$
- $N = \{S, NP, VP, N, V, Det\}$
- $P = \{S \rightarrow NP\ VP, NP \rightarrow Det N, VP \rightarrow V, Det \rightarrow$the, $N \rightarrow$dog, $V \rightarrow$barks$\}$

This is a typical example of a fragment of a context-free grammar used for parsing a natural language, in this case English. Of course, this is an extremely simplified "toy" grammar whose associated language $L(G)$ contains a single sentence ("the dog barks"); nevertheless, the larger grammars used in real-life applications often have the same structure: nonterminal symbols correspond to syntactic structures — such as noun phrases (NP) or verb phrases (VP) — and production rules express the valid ways in which these structures can combine into larger ones. Terminal symbols can denote concrete words, as in this example, or part-of-speech tags (such as N or Det). The latter option is common when a parser is used as one of the steps in a natural language processing pipeline, receiving its input from a *part-of-speech tagger* module which maps the words in input sentences to these tags.

We can check that the sentence "the dog barks" is in fact in $L(G)$ by obtaining it as a result of applying a sequence of productions to the initial symbol S, as explained before:
$S \Rightarrow NP\,VP \Rightarrow Det\,N\,VP \Rightarrow Det\,N V \Rightarrow$the $N V \Rightarrow$the dog $V \Rightarrow$the dog barks.

If we represent the derivations we have just made as a tree, where each application of a rule $A \rightarrow \alpha$ is represented by adding nodes labelled with the symbols in α as children of the node A, we obtain a *parse tree* for the sentence, which is shown in figure 3a.

The process of determining whether a given string $w_1 \ldots w_n$ belongs to the language defined by a grammar G by finding a sequence of derivations for it (or ensuring that none exists) is called *recognition*. The process of finding all the possible parse trees for the string $w_1 \ldots w_n$ is called *parsing*. Note that, in more complex grammars than this, there may be several different valid parse trees for a single sentence. For example, in a larger natural language grammar, the parsing of the sentence "John saw a man with a telescope" would result in two different trees (shown in figure 3b and 3c), the former corresponding to the interpretation "a man having a telescope was seen by John", and the latter corresponding to "John used a telescope to see a man". In this case we say that the sentence is *ambiguous*, and parsing it correctly implies finding the parse trees for every possible interpretation.

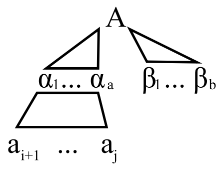

Figure 4. Form of the trees associated to the Earley item $[A \to \alpha.\beta, i, j]$.

2.2. Parsing schemata: an example

Parsing schemata, introduced by [28], 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 (*deduction 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.

We can understand how parsing schemata work by studying the semantics of the Earley schema shown in figure 2 (for a more formal explanation, see [28]).

Items in the Earley parser are tuples of the form $[A \to \alpha.\beta, i, j]$, where $A \to \alpha.\beta$ is a grammar rule with a special marker (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 follows: "There exists a valid parse tree with root labelled A, where the direct children of A are labelled with the symbols in the string $\alpha\beta$, the leaf nodes of the subtrees rooted at the nodes labelled α form the substring $a_{i+1} \dots a_j$ of the input, and the nodes labelled β are leaves". Such a tree can be seen in figure 4. Note that this item format and semantics is linked to the top-down, left-to-right strategy that the Earley parser uses to find parse trees, so schemata for different parsers will use different kinds of items.

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

A deduction step $\frac{\eta_1 \dots \eta_m}{\xi}$ Φ allows 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 as in this example or specify other constraints that must be verified in order to infer the consequent.

In this particular case, the *Initter* and *Predictor* steps are used to initialize the analysis by generating items with the dot in the first position of their associated production's right-hand side. These items represent the application of a production without as yet having recognized any input symbols. As we have seen, the dot in productions marks the region of their right-hand side which has been recognized, and the *Scanner* and *Completer* steps allow us to enlarge this region by shifting the dot to the right. The

Scanner step reads and recognizes a single terminal symbol from the input, while *Completer* recognizes a nonterminal symbol and joins two partial parse trees into a larger one.

If we now look at Earley's algorithm as described in figure 1, we can see that it is nothing but a particular implementation of this schema. The deduction of an item $[A \rightarrow \alpha.\beta, i, j]$ in the schema is implemented by adding a state $(A \rightarrow \alpha\beta, |\alpha|, i)$ to the set `S[j]`, and the *Predictor*, *Completer* and *Scanner* operations in the code correspond to the deduction steps in the schema. The loop and the structure used to hold the state sets impose a particular order on the execution of these operations, which guarantees that the state $(S \rightarrow \gamma, |\gamma|, 0)$ will be generated if the input is a valid sentence according to the grammar.

3. SYSTEM OVERVIEW

3.1. Motivation for our system

Parsing schemata are located at a higher abstraction level than algorithms. As we have just seen in the example, a schemata specifies a set of steps that must be executed and a set of intermediate results that must be obtained when parsing sentences, 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 test a parser in practice by running it on a computer, we need to implement it in a programming language, so we have to abandon the high abstraction level 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 ]
```

Note that while an implementation of the Earley parser in a programming language would probably take several hundred lines of code, the parsing schemata used by our system are compact, declarative, highly readable and easy to understand and modify.

3.2. Goals

Three main design goals have been taken into account during the development of our system:

- *Declarativity:* The input format for representing schemata to be compiled by our system should be highly declarative, and similar to the formal notation used to represent schemata. The system should take care of all the operations needed to transform this formal, abstract notation into a functional implementation of the corresponding parser. This allows the parser designer to focus on the semantics of the schema while abstracting from any implementation detail.
- *Generality:* The system should be able to handle all kinds of parsing schemata for context-free grammars and other formalisms. Note that this requirement is not trivial, since the formal notation for parsing schemata is open, so that any mathematical object could potentially appear in a schema.
- *Efficiency:* Implementations generated by the system should be efficient. Of course, we cannot expect the generated parsers to be as efficient in terms of runtime or memory usage as ad hoc implementations programmed by hand, but they should at least be equivalent in terms of computational complexity.

The declarativity goal has been achieved by defining a simple language to represent schemata, practically equal to the formal notation normally used in the literature, and using it as a starting point to generate Java code, which can in turn be compiled. Therefore, our system works in a similar fashion to parser generators such as Yacc [17] or JavaCC [30].

The generality goal has been achieved by means of an extensibility mechanism: since it would be impossible to support by default all the different kinds of object that could appear in schemata, we allow the user to easily define and add new object types, which can be handled by the code generator in the same way as the predefined ones.

Finally, the efficiency goal has been achieved by having our system perform a static analysis of input schemata in order to determine the data structures and indexes needed to provide constant-time access to items, and generate code for these indexes.

3.3. Related work

Although some previous work has been done on systems and techniques that can be used to implement parsing schemata for natural languages, the existing alternatives do not fulfill the features enumerated in section 3.2. The Dyna language [11] can be used to implement some kinds of dynamic programs, but its notation is not as close to the formal notation commonly used to represent schemata as ours. The DyALog system [8] can be used to compile and run tabular parsers for several grammatical formalisms, but the specifications are based on logical push-down automata and can be complex and unnatural, especially for purely bottom-up parsers which do not use a left-to-right strategy to process the input string. None of these systems is specifically oriented to the implementation of parsing schemata.

Shieber [27] introduces a technique to execute parsing schemata with a deductive parsing engine programmed in Prolog. However, this requires the conversion of items and deduction steps to the Prolog language. Moreover, if we want the implementations generated with this technique to be efficient, we need to provide item indexing code by hand, so we have to abandon the abstraction level of schemata and take implementation details into account. Without this indexing system, the Prolog interpreter will perform a large amount of CALL and REDO operations, distorting the results when working with large grammars [3, 9].

Basic parsing schemata can also be implemented in Datalog, a variant of Prolog commonly used for deductive databases. The subject of obtaining efficient implementations of Datalog programs has been studied in the literature [21, 20]. However, the constraints imposed by Datalog disallow some useful extensions to parsing schemata, like feature structure unification, that can be used in our system.

3.4. System architecture

Our parsing schemata compiler is composed of several different subsystems:

- The “sparser” (schema parser) subsystem reads input parsing schemata, parses them and transforms them into an internal tree representation, which will be the input to the code generation step. This subsystem is a compiler generated by the JavaCC parser generator.
- The “generator” (code generator) subsystem is the most complex part of the schema compiler. This subsystem takes the tree representation produced by “sparser” as input, and uses it to generate the Java classes implementing the algorithm described by the schema. This subsystem is divided into several parts, and each of them is used to generate a part of the implementation: deduction step execution, item management, indexing, etc.
- The “eparser” (element parser) subsystem guarantees the *generality* property discussed previously by providing an extensibility mechanism which can be used to compile schemata with non-predefined elements. As explained above, parsing schemata have an open notation, so any mathematical object could appear as part of an item. Therefore, it would be impossible for our system to recognize “a priori” any kind of item that could potentially appear in an arbitrary schema. The “eparser” subsystem allows the user to define his own kinds of notational elements and use them in schemata, the use of Java’s dynamic class loading facilities eliminating the need for recompilation in order to add new element types.

Figure 5 shows how these systems interact to transform a parsing schemata into an executable parser. More details about each of the subsystems will be given in the following sections.

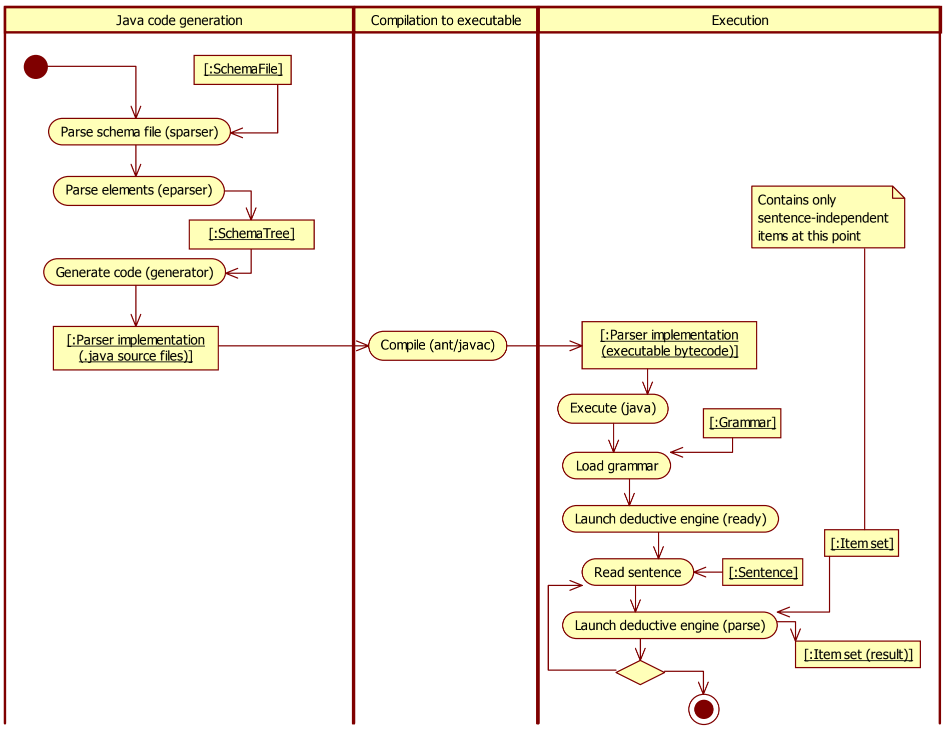
Java code generation
Compilation to executable
Execution
[:SchemaFile]
Parse schema file (sparser)
Parse elements (eparser)
[:SchemaTree]
Generate code (generator)
[:Parser implementation (.java source files)]
Compile (ant/javac)
[:Parser implementation (executable bytecode)]
Contains only sentence-independent items at this point
Execute (java)
[:Grammar]
Load grammar
Launch deductive engine (ready)
[:Item set]
Read sentence
[:Sentence]
Launch deductive engine (parse)
[:Item set (result)]

Figure 5. Activity diagram showing how the system can be used to compile and execute a parsing schema.

4. ARCHITECTURE OF THE GENERATED CODE

Before going into detail about the design of our code-generating system, we first need to think about the design of the code it will have to generate. The structure of this code must be generic enough to be applicable to any schema, but it must also allow us to include particular optimizations for each schema, to enable us to achieve the efficiency goal.

A deductive parsing engine such as the one described by Shieber in [27] fulfills the first condition, providing a generic means of implementing any parsing schema; but it is not efficient unless we can access items in constant time, and the way to achieve this is different in each particular schema. The idea of compiling parsing schemata allows us to generate schema-specific code to attain efficiency.

In particular, our compilation process proceeds according to the following principles:

- A class is generated for each deduction step. The classes for deduction steps implement a common interface with an `apply` method which tries to apply the 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.

- Code is generated to read an input grammar and create an instance of a deduction step class for each possible set of values satisfying its side conditions. For example, a distinct instance of the Earley Predictor step will be created at runtime for each grammar rule of the form $B \rightarrow \gamma \in P$, which is specified in the step's side condition. Deduction step instances are lightweight objects, so large grammars needing a large amount of them can be handled.
- The execution of deduction steps in the generated code is coordinated by a deductive parsing engine, which can be seen as an implementation of the dynamic programming approach that underlies chart parsing [19]. Since this is a generic algorithm, the parsing engine will always be the same and we do not need to generate it. The engine works as described by the following pseudocode:

```
steps = set {deduction step instances};
items = set {initial items};
agenda = list [initial items];
For each deduction 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 deduction 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 deduction steps) and an agenda, implemented as a queue, containing the items with which we have not yet tried to trigger new deductions. 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.

- An `ItemHandler` class is generated to provide efficient access to items. This class contains indexing code specific to each schema, since the best choice of indexes will depend on the particular features of each. Additionally, a `StepHandler` class is generated to provide efficient access to deduction steps.

Schema ::= [ElementDefinitionList] [OptionList]
  { StepName StepDescription } { **@goal** GoalDescription }
ElementDefinitionList ::=
  @begin_elements { ElementDefinition } **@end_elements**
ElementDefinition ::= $element\_definition$
OptionList ::= { **@begin_options** Option **@end_options** }
Option ::= **@option** key value
StepName ::= **@step ID**
StepDescription ::= Antecedent Separator Conditions Consequent
GoalDescription ::= Antecedent
Antecedent ::= { ItemDescription }
Separator ::= { ”–” }
Consequent ::= ItemDescription
ItemDescription ::= ”[” ElementList ”]”
ElementList ::= [ElementWrapper { , ElementWrapper }]
ElementWrapper ::= Element
Conditions ::= ElementList
Element ::= $element$

Figure 6. EBNF grammar for parsing schema files.

5. READING SCHEMATA

As we have explained above, the goal of the “sparser” subsystem is reading an input file with the description of a parsing schema and converting it to an internal tree representation holding the data that will be passed to the next subsystem, the code generator. The notation used to describe schemata is very simple, and practically identical to the formal notation commonly used to define them. More concretely, the schema file format is the one described by the EBNF grammar in figure 5.

As we can see, there are two symbols in the EBNF grammar (*element* and *element_definition*) which are undefined. This is because their definition will vary depending on the custom notational elements defined by the user. Actually, from the point of view of the “sparser”, the definition of these symbols is a generic regular expression accepting any string without spaces or commas which cannot be confused with other components of the schema file. When the “sparser” finds one of these strings in a position where an *element* or *element_definition* is expected, it will delegate its analysis to the “eparser” module, which deals with elements and element definitions. In the remainder of this article, we will use the word *element* to refer to any object that can appear as part of an item.

The general structure of a parsing schema file consists of an optional section with *element definitions*, a second optional section containing *options*, a series of *deduction steps*, and a series of *goals* or final items. An example of a schema file containing all these sections is the following:

```
@begin_elements
element.Symbol:nonGroundFromString:[A-RT-Za-ho-z]
element.Symbol:groundFromString:S
element.RuleWrapper:fromString:[A-Za-z \.]+->[A-Za-z \.]*
element.StringPosition:nonGroundFromString:[i-n]
element.StringPosition:groundFromString:[0-9]+
element.SumOfPositionsExpression:fromString:[0-9i-k\+\-]+
element.SymbolSequence:fromString:((alpha)|(beta)|(gamma))
element.SpecialElement:fromString:\.
@end_elements

@begin_options
@option outputItems allItems
@end_options

@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 ]

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

As we can see, the element definition section is used to define the types of element that will appear in the schema's deduction steps. Element definitions map regular expressions to Java classes and methods. For example, the element definition

```
element.StringPosition:nonGroundFromString:[i-n]
```

means that, whenever a lowercase letter in the range $i \ldots n$ is found in an item, an instance of the `StringPosition` class must be created by invoking the method with signature `public static StringPosition nonGroundFromString(String s)` in the `element.StringPosition` class. This job is done by the "eparser", as mentioned earlier. Note that the reference to "non-ground" in the method name means that the generated instance will represent a variable. Constant (ground) string positions are defined in the example by a different regular expression (`[0-9]+`).

In this case, element definitions are shown only for explanatory purposes: all the elements used in the Earley schema are already predefined in the system, so we do not need to explicitly redefine them. Explicit definitions are only needed to include new kinds of elements defined by the user, or for overriding the default regular expressions associated with the predefined elements.

The options section is used to parametrize the resulting parser. In this example, we pass an option to the system indicating that we want the generated parser to output all items obtained for each sentence (if no option were used, only the goal items would be output). Options can also be used to dynamically change the type of agenda or deductive engine: for example, for error-correcting parsing, we could need an agenda implemented as a priority queue instead of a standard queue, so that the items with smaller error count could be used first. In order to use such an agenda, we would use a line

```
@option agendaClass agenda.PriorityQueueAgenda
```

and define an `agenda.PriorityQueueAgenda` class implementing a simple `Agenda` interface. The content of `@option` lines is also accessible via a simple API from the generated code, so that user-defined classes such as this agenda can also use `@option` lines for further parametrization.

After these optional sections, we define the deduction steps of our schema in the simple notation mentioned in section 3.1, and then specify the format of the final items with one or more `@goal` lines. If items matching a `@goal` specification are found by the generated parser, the parsing process is considered to have been successful and these final items are output.

In order to implement the "sparser" subsystem, the JavaCC [30] compiler compiler has been used. This tool generates an LL(k) compiler from a grammar annotated with Java code. In this case the code is simple, since it only has to build a tree which will be passed as input to the code generator. One of the advantages of using an LL(k)-based compiler compiler such as JavaCC is that it provides helpful error messages by default, thus making it easy to locate syntax errors in parsing schema files.

The tree produced by the "sparser" is nothing more than a hierarchical representation of the schema, where the schema itself, deduction steps, antecedents, etc. are represented by tree nodes. The leaf nodes in this tree are the components of items that we have called *elements*, and are instantiated by the "eparser".

6. CODE GENERATION

The "generator" subsystem is the most complex and important component of the parsing schema compiler. From a tree representation of a parsing schema generated by the "sparser" and "eparser", this component generates Java code for the classes implementing the corresponding algorithm. For the sake of simplicity, we will use a parsing schema corresponding to the CYK [18, 31] bottom-up parser in some of the code generation examples. This schema is one of the simplest that we can find in practice, having fewer steps and fewer kinds of element than Earley's, and can be defined as follows [28]:

```
@step D1
[ a , i , i+1 ]
--------------------- A -> a
[ A , i , i+1 ]
@step D2
[ B , i , j ]
[ C , j , k ]
--------------------- A -> B C
[ A , i , k ]
@goal [S,0,length]
```

6.1. Types of element

As we have seen in the previous section, the leaf nodes of a schema tree contain the basic components of items, called elements. Since we want our system to be able to work with all kinds of parsing schemata, and any mathematical object could potentially appear in the representation of an item, we have implemented an extensibility mechanism that allows the user to define custom elements if the predefined element classes do not suffice to represent a particular schema. This extensibility mechanism works by allowing the user to define regular expressions to represent new kinds of element, and associate them to classes. The problem is that the code generator should be able to handle these user-defined elements and use them successfully to generate efficient code. In order to achieve this, our system requires element classes to follow a simple contract, providing the services needed by the code generator. This basic contract comes from the idea that any element appearing in a schema can be classified into one of four basic 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, etc.
  In order to define a new simple element class, the user must implement a `SimpleElement` interface, providing a method to obtain a Java code representation of the element's value, if it has one. For example, the Java code representation of an element representing a string position and holding the value 1 is the string "StringPosition.groundFromValue(1)", which calls a static method returning an integer element with the value 1[†]. The method returning the Java code representation for grammar symbols is as follows:

[†]Using a static method such as this one instead of creating a new instance ("new StringPosition(1)") is an optimization. A parser may use millions of items, each of them with several elements, so all the predefined element classes are programmed in such a way that the generated code uses multiple references to the same instances instead of multiple instances. Apart from the memory saved with this optimization, it must also be noted that item comparison is one of the main performance bottlenecks in generated parsers, and this optimization allows such comparisons to be performed at the reference level, which is much faster than dereferencing the elements and comparing their values.

```
public String getExpressionCode()
{
        return "element.SymbolTable.instance ().
        symbolFor(\""+SymbolTable.instance (). getName(this)+"\")";
}
```

In addition to this method, simple element classes should implement a method returning an integer indexing key, so that the corresponding elements can be used for item indexing.

- *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 addition of two string positions. Feature structures and logic terms are also represented by this kind of element. 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. Expressions have been used to implement unification of feature structures [5], left-corner relationships [23], etc.
  When defining a class for a new expression, the user must implement an `ExpressionElement` interface, providing a method to obtain a Java code representation of the expression from the representation of its children (operands). For example, the method for sums of string positions takes an array of strings as a parameter ($[op_1, op_2, \ldots, op_n]$) and returns the string "$op_1 + op_2 + \ldots + op_n$". Apart from this, the user must also provide a method returning the return type of the expression: in this case, the class `element.StringPosition.class`.
- *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 $[A \to \alpha.B\beta, i, j]$ is represented as a composite element with three components, the first being in turn a composite element representing a grammar rule.
  The interface for this kind of element, `CompositeElement`, only requires the user to provide methods returning the number of children (sub-elements) of a composite, and to get the *i*th child.
- *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 interface `SequenceElement` only requires the user to provide a method returning a type for the elements in the sequence. For example, the class representing symbol sequences has a method that always returns `element.Symbol.class`. It is possible to define sequences holding elements of multiple types by returning a more generic type such as `element.SimpleElement.class`.

6.2. Deduction step classes

Each of the deduction steps in the schema, represented by `@step` specifications in the input file, produces a class implementing the `DeductionStep` interface. Goal specifications `@goal` also produce deduction step classes, as if they were steps with a single antecedent and no consequent, since in this way the indexing and matching techniques used to find items matching antecedents can be reused to find the goal items in an item set.

The main function of deduction step classes is to provide a method that, given a particular item, generates all the items that the step can deduce using that item as an antecedent, and previously generated items for the rest of the antecedents. This functionality is provided by a

```
List applyTo ( Object[] item )
```

method in the `DeductionStep` interface, which will be implemented by each concrete deduction step class created by the code generator.

6.3. Representation of items in the generated code

As can be seen in the signature of the `applyTo` method, items in the generated code are represented as object arrays (`Object[]`). This may come as a surprise since, when we described the types of element in section 6.1, we mentioned that items were represented by instances of the class `CompositeElement`.

The reason for this discrepancy is that the representation of elements and items in the generated code is not the same as that handled by the code generator. In the code generator, elements of schemata are represented by a hierarchy with its base in the `Element` class, and the *Composite* design pattern [12] is used to represent items as tree structures. This way of modelling elements is elegant from an object-oriented design standpoint, makes it easy for the user to add custom element types and simplifies system maintenance.

In the generated code we also need to work with elements and items, but priorities are different. Generated code is a "black box" that does not need maintenance by the user (modifications in generated parsers should be made by modifying the input schema and regenerating the code). Taking this into account, it is reasonable to prioritize efficiency over elegance in the generated code. This is the reason why, in the generated code, composite element structures such as items are translated to object arrays, whose components can be elements or other object arrays. This makes generated code somewhat convoluted and hard to read, but more efficient, since the array representation of items saves indirection levels and memory usage with respect to a more object-oriented representation. As an example, we need $32 + 3S$ bytes to represent the item $[A, 0, 2]$ in the code generator, where S is the object shell size[‡], and `ArrayList`s are used to implement composites. With the array representation used in the generated code, the same item takes up only $16 + S$ bytes. Since natural language parsers typically need to store hundreds of thousands of items, this difference in memory usage is important, and the elimination of one indirection level also affects parser runtimes.

[‡]The *object shell size* is the minimum object size in a Java Virtual Machine (JVM). The concrete value of S depends on the particular JVM used to execute Java code, but is typically 8 bytes in modern JVM's.

While composite elements are represented by arrays in the generated code, expression and sequence elements have no particular representation: these elements are transformed into operations rather than data structures. Expression elements appear in the generated code as Java expressions producing a simple element result; and sequence elements will produce code to match zero or more simple elements.

6.4. Visitors for code generation

The two most complex methods in a deduction step class are the constructor and the aforementioned `applyTo` method. If a deduction step has a production rule as a side condition, the constructor must check if a rule passed as a parameter matches the condition, and initialize the corresponding variables. Therefore, the constructor of the step

```
@step D2
[ B , i , j ]
[ C , j , k ]
--------------------- A -> B C
[ A , i , k ]
```

will check whether the parameter is an array of length 3, and initialize the variables A, B and C in the step to the concrete values found in the parameter. Therefore, if the initialization parameter is the concrete rule $S \rightarrow NP\,VP$, the constructor will assign $A = S$, $B = NP$ and $C = VP$.

On the other hand, the `applyTo` method returns all the items which can be generated using the one passed as a parameter as an antecedent, and previously generated items for the rest of the antecedents. As an example, suppose that we have the instance of the step created with the rule $S \rightarrow NP\,VP$, and we receive the item $[NP, 0, 2]$ as a parameter. The operations needed to implement the `applyTo` method are the following:

- Match the given item with the specification $[B, i, j]$ where the value of B must be NP and i, j can take any value.
- If it matches, assign particular values to the variables i and j (in this case, the matching is successful, and we assign $i = 0$ and $j = 2$).
- Search for *all* the items in the item set that are of the form $[C, j, k]$ where the value of C is VP and the value of j is 2. That is, search for the items of the form $[VP, 2, ?]$.
- For each of these, generate a conclusion item $[S, 0, k]$ with the corresponding value of k.
- Repeat all the steps for the other antecedent, i.e., match the given item with the specification $[C, j, k]$ and then search for items verifying $[B, i, j]$. In our particular case, the item $[S, 0, 2]$ does not match the second antecedent.

Putting it all together, in order to generate code for the constructor and `applyTo` methods, we need a way to obtain code for the following operations:

- Match a given item with a specification. The specification may come from an antecedent or a side condition, and is known at schema compile time, while the item is only known at runtime.
- Search for all items matching a specification known at compile time.
- Use a specification to initialize step variables to values taken from an item.
- Generate a conclusion item from step attribute values.

The code for all these operations can be generated in a similar way if we take into account that all of them traverse an item and are directed by a specification. We have used the Visitor design pattern [12] to structure this part of the code generator. Code generating visitors traverse the parts of the schema tree corresponding to item specifications. There is a different visitor for each basic operation in the generated code (matching, assigning values, etc.) and each visitor has a different behaviour for each kind of node (`SimpleElement`, `ExpressionElement`...) in the specification. We also need to keep track of which variables in specifications have a concrete value at each part of the code and which are uninstantiated, so this information is kept by an external structure which can be queried by the visitor.

The visitors themselves are also stateful, since they keep code for accessing parts of items as an internal state. This is because some information generated when matching an element can be needed to generate the matching code for subsequent elements. This can be seen in this sample of generated code[§], which checks whether an item in the Earley algorithm conforms to a generic specification $[A \to \alpha.B\beta, i, j]$ (where the variables are not yet instantiated and could take any value):

```
// structural check
if ( ((Object[]) item).length != 3 ) return result;
if ( item[0] instanceof Object[] )
{
        if ( ((Object[]) item[0]).length < 3 ) return result;
}
else
        return result; // matching failed
// "matching" with trigger item
if ( ((Object[]) item[0])[0] instanceof element.Symbol ) // class check
        sp_A = ( element.Symbol ) ((Object[]) item[0])[0];
else
        return result; // matching failed
int sp_alpha_index = 1; // variable to read symbols from the sequence alpha
while ( sp_alpha_index < ((Object[]) item[0]).length
  && ((Object[])item[0])[ sp_alpha_index ] instanceof element.Symbol )
{
        sp_alpha.add(((Object[]) item[0])[ sp_alpha_index ]);
        sp_alpha_index++;
}
```

[§]The generated code is not shown literally. It has been simplified by removing some optimizations in order to make the example more compact and readable.

```
if ( item[0] instanceof Object[] )
{
        if ( ((Object[]) item [0]). length < 2+sp_alpha_index )
                return result ; //matching failed
}
else
        return result ; //matching failed
if ( !((Object[]) item[0])[0+ sp_alpha_index ]. equals(Dot. getInstance ()) )
        return result ; //matching failed
if ( ((Object[]) item[0])[1+ sp_alpha_index ] instanceof element.Symbol )
{
        sp_B = ( element.Symbol ) ((Object[]) item[0])[1+ sp_alpha_index ];
}
else
        return result ; //matching failed
int sp_beta_index = 2+sp_alpha_index;
 (...)
```

When the visitor that generates matching code visits the sequence element node corresponding to α, it inserts the declaration for a new variable `sp_alpha_index` into the code. This variable is used as a loop index when reading symbols in the sequence α, and its value at the end of the loop will depend on the number of symbols that match α for each particular item. This value must then be used to access the subsequent elements: for example, in order to access B in $[A \to \alpha.B\beta, i, j]$ the code `((Object[])item[0])[1+sp_alpha_index]` is used. A representation of the code being used to access the item is stored in the visitor's state so that this information can be kept between invocations.

6.5. Search specifications

In the previous section we mentioned that one of the operations the `applyTo` method needs to perform is to search for all the items matching a specification known at compile time. While the rest of the operations that we have mentioned work on a single item, this one must access the item set. This operation is not really implemented by the deduction step classes, but in an `ItemHandler` class that provides efficient access to items by using indexes specifically generated for each schema.

The `ItemHandler` class provides three services: adding an item to the item set including it in the corresponding indexes, checking whether a given item is present or not in the item set, and returning all items verifying certain characteristics. All of these methods need indexing techniques in order to work efficiently.

In order to call the third method, which is the one used by `applyTo` to search for antecedent items, we need a way of specifying constraints on items. A simple and efficient way to do this is by representing search constraints in the same way as items, but using `null` values to represent unconstrained elements. Therefore, in our example where the CYK `D2` step needed to search for items of the form $[VP, 2, ?]$, the deduction step class would pass the specification $[VP, 2, null]$ to the item handler class. The `ItemHandler` will then return all items of this form by using its indexes.

```
Object[] sp_skeleton_SP_CYKMainStep0_1 = new Object[] { sp__C, sp_i2 , null };
 List items1 =
ItemHandler. instance (). getBySpecification ( sp_skeleton_SP_CYKMainStep0_1 );
```

7. INDEXING

If we wish our generated parsers to achieve the efficiency goal mentioned in section 3.2, access to items and deduction steps must be efficient. As we have seen in the previous section, when we execute a step we often need to search the item set for all the items verifying a given specification. 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.

As an example, the theoretical complexity of the CYK parsing algorithm is $O(n^3)$, where n is the length of the input. This is because the most complex step in this algorithm is

```
@step D2
[ B , i , j ]
[ C , j , k ]
--------------------- A -> B C
[ A , i , k ]
```

which can be executed on at most $O(n^3)$ combinations of antecedents, since positions i, j and k take values between 0 and n and symbols A, B, C come from a finite set.

As we have seen, the `applyTo` method that executes this step in the generated code matches the item received as a parameter with the specification $[B, i, j]$ and then searches for all items in the item set of the form $[C, j, k]$ for fixed values of C and j. If we can obtain a list of these items in constant time, the `applyTo` method will run in $O(n)$[¶] (since we have to traverse this list and generate a conclusion for each of the items), and it will generate $O(n)$ items. Since the total number of items generated in a CYK parser is $O(n^2)$ (items have two indexes ranging from 0 to n), this `applyTo` method will be invoked $O(n^2)$ times during the execution of the parser. Therefore, the total complexity is $O(n^2) \times O(n) = O(n^3)$, matching the theoretical computational complexity of CYK. However, if we had no indexation and the search for items were sequential, the `applyTo` method would run in $O(n^2)$ (there are $O(n^2)$ items to search among) and the generated implementation for CYK would be $O(n^4)$.

[¶]In this reasoning about complexity, we are only taking into account the first part of the `applyTo` method, which matches the parameter item with the first specification and then searches for items conforming to the second. However, if we apply an analogous reasoning to the second part of the method (i.e. applying matching to the second specification and searching to the first), we obtain that the second part is also $O(n)$, so the method is globally $O(n)$.

7.1. Static analysis and index descriptors

Generating indexes that can provide constant-time access to items is not a trivial task, since a generic indexing technique does not suffice: the elements by which we should index items in order to achieve efficiency vary among schemata. For example, the CYK parser's deduction steps perform two different kinds of searches for items: searches for items of the form $[C, j, ?]$ (where $?$ can take any value) and searches for items of the form $[B, ?, j]$. Thus, in order to ensure that these searches access items in constant time, we need at least two indexes: one by the first and second components and another one by the first and third. Different parsing schemata, as well as different steps in the same schema, will have different needs. Therefore, in order to generate indexing code, we must take the distinct features of each schema into account.

The decision of which indexes to create for a given schema is taken by performing a static analysis of each deduction step in order to determine the kind of searches it needs to perform. This information is known at schema compilation time and is gathered by our system during deduction step code generation, stored in data structures called *search descriptors*. For example, when the code generator produces the code for this search in the CYK parser:

```
Object[] sp_skeleton_SP_CYKMainStep0_1 = new Object[] { sp__C, sp_i2 , null };
List items1 =
ItemHandler. instance (). getBySpecification ( sp_skeleton_SP_CYKMainStep0_1 );
```

the code-generating visitors, apart from outputting the search specification `new Object[] { sp__C, sp_i2, null }`, also produce a tree structure of the form `[ Symbol , StringPosition , null ]`. This structure, called a *search descriptor*, specifies the structure of the items that are searched for and the positions and classes of elements which take concrete values in the search specification.

Search descriptors from all the deduction steps in the input schema are gathered into a list, and used to decide which indexes to create. It will be convenient to create indexes by non-null components of search descriptors that can be used for indexing (i.e. belonging to a class that provides a method to obtain an indexing key, see section 6.1). The simplest way to do this is by creating an index for every search descriptor, indexing by all components meeting these conditions. With this approach, the presence of our search descriptor `[ Symbol , StringPosition , null ]` means that we should generate an index on the first and second components of items, and the other search descriptor obtained from the same step (which is `[ Symbol , null , StringPosition ]`) means that we should generate an index on the first and third component.

The decisions that the system takes about the indexes it needs to create are encoded into objects called *index descriptors*, which are lists containing the positions of elements used for indexing and the type of indexes that are going to be used. For example, an index descriptor for our first index in this case could be `[0:hash,1:hash]`, meaning that we are going to use the elements in positions 0 and 1 as keys for hash indexes[‖]. The decision as to which particular data structures to use for indexes

[‖]Note that items are trees, not lists, so in a general case the position of an element cannot be denoted by a single integer. Positions are represented by lists of integers: for example, when working with Earley items of the form $[A \to \alpha.B\beta, i, j]$, we can have a (hashes, arrays...) can be configured as an option, either by setting a global default or by configuring it for each particular element class.

7.2. Generation of indexing code

Once we have index descriptors for all the search indexes we will need, we can proceed to generate indexing code. This code is located in the `ItemHandler` class which, as mentioned in section 6.5, provides three services: finding all items verifying a given specification (`getBySpecification`), checking whether a given item exists in the set (`exists`) and adding an item to the set (`add`).

The `getBySpecification` service uses *search indexes*, which are obtained from index descriptors obtained as described in the previous section. The `exists` service uses a different kind of indexes called *existence indexes*. These are obtained in the same way as search indexes, but their search descriptors come from a full consequent item instead of from a search specification, and have no null values. The `add` service must use both search indexes and existence indexes, since every item added to the set must be accessible to the other two services.

Although the functionality of each of the three services is different, their implementation can be done in such a way that a significant part of the code is common to all of them, and we can take advantage of this fact during code generation. In particular, we can describe the three methods with the following high-level pseudocode:

```
method ( item or specification )
{
        test whether parameter conforms to search descriptor associated to index 1;
        **if** it does
        {
                access index 1 using parameter;
                process obtained list ;
        }
        (...)
        test whether parameter conforms to search descriptor associated to index d;
        **if** it does
        {
                access index d using parameter;
                process obtained list ;
        }
}
```

Note that, although we mention search descriptors in the pseudocode, search descriptors are not accessible from the code, they are only used to generate it. Each of the d tests in the pseudocode corresponds to a different series of conditional statements that check if the parameter conforms to the

search descriptor `[[null,null,null,Symbol,null],null,IntElement]`, and the corresponding index descriptor for an index by B and j would be `[[0,3]:hash,[3]:hash]`.

structure expressed by a search descriptor, but they do not use the descriptor itself: its constraints are directly compiled into code. Also note that, although the conditions and bodies of the `if` statements are expressed in an uniform way in the pseudocode, they are different in the code, since they are generated from different search descriptors. This is the reason why we have expressed the pseudocode as a series of conditional statements, and not as a loop.

The main conceptual difference between the three methods is the meaning of “process obtained list”. In the case of `add`, processing the list means initializing it if it is null and adding the parameter item to it. In the case of `exists`, it consists of checking if the list is empty. Finally, in the case of `getBySpecification`, the method will simply return the obtained list.

In reality, the parts appearing as common in the pseudocode are also slightly different, but the differences are small enough to allow us to reuse most of the generator code.

The strategy for generating the code for these methods is similar to the one used in step classes. In this case, instead of traversing an element tree, we traverse a search descriptor, generating code at each node. We do not use the Visitor design pattern because the behaviour at each node depends on its content, not its class.

A high level pseudocode for generating the code to test whether a parameter conforms to the search descriptor is the following:

```
generateIndexingCheckCode ( SearchDescriptorNode node , List address , String objectName )
{
  if ( node is associated to a class )
    objectName = ”(Class)” + objectName;

  add (address,objectName) to address map;

  if ( node is not associated to a class )
    return ””; // null node => this part of items is not used in the indexing code

      if ( node associated to a class other than SequenceElement )
        generate type check:
          ”(object in ObjectName) instanceof (class associated to node)”

  if ( node is associated to class SequenceElement )
  {
    if ( operation = getBySpecification ) //parameter is a specification
      generate type check:
        ”(object in ObjectName) instanceof SequenceElement”
    if ( operation = exists or add ) //parameter is a concrete item
    {
      Class cl = expected class for members of the sequence;
      if objectName is of the form ”(∗)[i]”
      {
        generate loop:
        ” int  newIndex = 0;
          while  (  i+newIndex < (∗). length  && (∗)[i+newIndex]  instanceof   cl  )
            newIndex++;”
        objectName =  (∗)[ i+newIndex];
      }
    }
  }

  if  (  node  is   associated   to   array   )
  {
    generate   length  checks;
    for  i  =  0.. numChildren()  −1
    {
      child  =  i+1th  child  of  node;
      address  =  add(address , i );
      update  objectName to   traverse   child ;
      doGenerateCode (  child   ,  address   ,  objectName );
      address  =  removeLast(address );
    }
  }
}
```

In this recursive method, `node` holds a particular node in a search descriptor that we are using to generate code, `address` holds its address expressed as a list of integers, and the string `objectName` stores the code that should be used to access the corresponding element in the generated code.

Apart from generating the checks needed to match an item or specification to a search descriptor, this method also introduces entries into an *address map*, as can be seen in the code. The address map is used to convert positions in a search descriptor (which are lists of integers) to the string used to access the corresponding parts of items and specifications in the generated code. For example, the following address map is produced when executing this code on a search specification for items of the form $[A \to \alpha.B\beta, i, j]$ in Earley's algorithm:

Address	Object name
$[]$	`((Object[])item)`
$[0]$	`((Object[])item)[0]`
$[0,0]$ (A)	`((Object[])((Object[])item)[0])[0]`
$[0,1]$ (α)	`((Object[])((Object[])item)[0])[1]`
$[0,2]$ (.)	`((Object[])((Object[])item)[0])[1+ind4]`
$[0,3]$ (B)	`((Object[])((Object[])item)[0])[2+ind4]`
$[0,4]$ (β)	`((Object[])((Object[])item)[0])[3+ind4]`
$[1]$ (i)	`((Object[])item)[1]`
$[2]$ (j)	`((Object[])item)[2]`

The information in the address map is then used to generate the code to actually access the indexes. The need for the address map arises because the values of loop indexes declared as part of the checking code (such as `ind4` in this case) will be used by the index access code.

Generation of the index access code has a quite complex implementation, as we support different indexing data structures including arrays and collections (such as hash maps), which are accessed through different Java syntax. We also must take into account that indexes can be nested, but the result of an intermediate query can be `null`. For example, if we use hash indexing with two components of items as keys, a hash map will be queried by using the first component, and the result of the query will be a second hash map that can be queried by using the first component. But the first query can also return `null`, and we have to check this condition to avoid trying the second query on a null object and causing an exception.

For space reasons, we will not go into further details on this part of the code generator. An example of the code generated for hash indexes, and using the address map shown before, is the following:

```
if  (  ( partial0 =((HashMap)(ex_index2).get(
        ((element.Symbol)((Object [])(( Object []) item  )[0])[0]).  getHashKey())
   ))  !=  null)
{
    result   =((HashMap)partial0.get(
           ((element.Symbol)((Object [])(( Object []) item )[0])[2+ ind4 ]). getHashKey()
       ));
}
```

7.3. Indexing deduction steps

Apart from the indexes on items explained above, our system also includes *deduction step indexes* in the generated parsers. These indexes are used to optimize the process of deciding which deduction step instances can be applicable to a given item. Instead of blindly trying to apply every step and let the pattern-matching processes discard those not matching the processed item, we use the index to obtain a set of potentially applicable step instances, the rest (which are known not to be useful) being directly discarded.

As particular instances of deduction steps in a schema are usually tied to grammar rules, deduction step indexes do not improve computational complexity with respect to string length (which is already optimized by item indexing), but they can improve complexity with respect to grammar size. This is usually an important factor for performance in natural language applications, since it is common to use grammars with thousands of rules.

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

8. EXPERIMENTAL RESULTS

As an example of the performance that can be achieved by the parsers generated by our system in real-life domains, we will show empirical results obtained by generating implementations of several different natural language parsing algorithms and applying them to natural-language grammars.

8.1. Context-free grammars

We have used our system to generate implementations of three popular algorithms for context-free grammars: CYK, Earley and Left-Corner [23]. 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 [4]. 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 store and use the item with the highest probability when several items differing only in their associated probabilities are found.

The three algorithms have been tested with sentences from three different natural language grammars: the English grammar from the Susanne corpus [25], the Alvey grammar [6] (which is also an English-language grammar) and the Deltra grammar [26], 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. 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 a large number of different 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. Note that the generation process was oriented to obtaining sentences of the lengths we wished to study, rather than to simulating the balance of sentence lengths found in naturally-occurring language use.

Table I shows performance results (in terms of runtime and amount of items generated) for all these algorithms and grammars. The tests were performed on a low-end 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.

The success of the index generation techniques described in this article is shown by the fact that the empirical computational complexity of the three parsers is below their worst-case complexity of $O(n^3)$. Additionally, the results of the test can be used to compare the algorithms and grammars in an homogeneous environment, drawing the following conclusions (described in more detail in [16]):

- By looking for functions $f(n)$ such that the sequences $T(n)/f(n)$ seem to converge to a positive constant ($T(n)$ being the average time elapsed by a parser when processing strings of length n) we can estimate computational complexities. This allows us to see that the empirical computational complexity is lower in the case of the Susanne grammar (where it is close to linear) than in the other two grammars. The Susanne grammar also provides the best parsing

Table I. Performance measurements for generated parsers.

Grammar	String length	Time Elapsed (s) CYK	Earley	LC	Items Generated 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

times in absolute terms. The probable reason for this is that the Alvey and Deltra grammars have more ambiguity, since they are designed to be used with feature structures, and information has been lost when these features were removed from them.

- CYK is the fastest algorithm in all cases, and generates fewer items than the others.
- Left-corner is notably faster than Earley in all cases, but the degree of improvement it provides depends on each particular grammar. The Susanne grammar seems to be particularly well suited for left-corner parsing.

8.2. Tree-adjoining grammars

Although all the examples we have seen so far correspond to context-free parsing, our system is not limited to working with context-free grammars, since parsing schemata can be used to represent parsers for other grammar formalisms as well. Different formalisms can be added by defining element classes for their rules using the extensibility mechanism explained in section 6.1.

In particular, we have also used our system to generate parsers for tree-adjoining grammars [29]. 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

Table II. Runtimes obtained by applying different TAG parsers to several sentences from the XTAG distribution.

Sentence	Runtime (s)			
	CYK	Ear. no VPP	Ear. VPP	Neder.
He was a cow	2.985	0.750	0.750	2.719
He loved himself	3.109	1.562	1.219	6.421
Go to your room	4.078	1.547	1.406	6.828
He is a real man	4.266	1.563	1.407	4.703
He was a real man	4.234	1.921	1.421	4.766
Who was at the door	4.485	1.813	1.562	7.782
He loved all cows	5.469	2.359	2.344	11.469
He called up her	7.828	4.906	3.563	15.532
He wanted to go to the city	10.047	4.422	4.016	18.969
That woman in the city contributed to this article	13.641	6.515	7.172	31.828
That people are not really amateurs at intelectual duelling	16.500	7.781	15.235	56.265
The index is intended to measure future economic performance	16.875	17.109	9.985	39.132
They expect him to cut costs throughout the organization	25.859	12.000	20.828	63.641
He will continue to place a huge burden on the city workers	54.578	35.829	57422	178.875
He could have been simply being a jerk	62.157	113.532	109.062	133.515
A few fast food outlets are giving it a try	269.187	3122.860	3315.359	

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 *auxiliary 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 four different parsers for tree-adjoining grammars [1, 2]: a CYK-based algorithm, two extensions of Earley's algorithm with and without the valid prefix property (VPP), and Nederhof's parsing algorithm. These implementations were tested with a real-life, wide-coverage feature-based tree-adjoining grammar: the XTAG English grammar [32].

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 elements 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 deduction steps. In order for the steps to be able to check whether an 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 grammar, we also need to include feature structures inside items and add unification operations to the deduction steps.

Table II contains a summary of the execution times obtained by our parsers for some sample sentences from the XTAG distribution. The runtimes are within the expected complexity bounds, but not as fast as the ones we would obtain if we used Sarkar's XTAG distribution parser written in C [24]. This is not surprising, since our parsers have been generated by a generic tool without knowledge of the grammar, while the XTAG parser has been designed specifically for optimal performance in this particular grammar and uses additional information (such as tree usage frequency data from several corpora, see [32]).

However, our comparison allows us to draw conclusions about which parsing algorithms are better suited for the XTAG grammar. In this case there is not a single best algorithm in terms of execution time, since the performance results depend on the size and complexity of the sentences. Therefore, in practical cases, we should take into account the most likely kind of sentences that will be passed to the parser in order to select the best algorithm. For example, we can see that the CYK parser has a poorer performance than others for short sentences, but is faster for longer sentences.

More detailed information on these experiments with the XTAG English grammar and other tree-adjoining grammars can be found at [14, 13].

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the design and implementation of a working compiler which is able to automatically transform formal specifications of parsing algorithms (expressed as parsing schemata) into efficient implementations of the corresponding parsers. The system's source code can be downloaded from `http://www.grupocole.org/software/COMPAS/`.

Our compiler takes a simple representation of a parsing schema as input and uses it to produce optimized Java code for the parsing algorithm it describes. The system performs a static analysis of the input schema in order to determine the adequate indexes and data structures that will provide constant-time access to items, ensuring the efficiency of the generated implementation.

The ability to easily produce parsers from 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. The implementations generated by our system are efficient enough to be used as prototypes in real-life domains, so they provide a quick means of evaluating several parsing algorithms in order to find the best one for a particular application. This is especially useful in practice, since different parsing algorithms can be better suited to different grammars and domains.

The system is general enough to be applicable to different grammatical formalisms, and has been used to generate parsers for context-free grammars and tree-adjoining grammars. In addition, we provide an extensibility mechanism that allows the user to add new kinds of elements to schemata apart from the predefined ones. This same mechanism has been used to provide predefined extensions like those for feature structure unification and probabilistic parsing.

Currently, we are using the system to automatically derive robust, error-correcting parsers from standard parsers for context-free grammars and tree-adjoining grammars. Additionally, we are working on its application to projective and nonprojective dependency-based parsing [22], since dependency parsers can also be represented by parsing schemata, as described in [15].

REFERENCES

1. Alonso MA, Cabrero D, Clergerie EV, Vilares M. Tabular algorithms for TAG parsing. In *Proc. of the 9th Conference of the European Chapter of the Association for Computational Linguistics (EACL'99)*, pages 150–157, 1999.
2. Alonso MA, Clergerie EV, Díaz VJ, Vilares M. Relating tabular parsing algorithms for LIG and TAG. In *New Developments in Parsing Technology*, pages 157–184, Kluwer Academic Publishers, Dordrecht-Boston-London, 2004.
3. Alonso MA, Díaz VJ. Variants of mixed parsing of TAG and TIG. In *Traitement Automatique des Langues,* 44(3):41–65, 2003.
4. Billot S, Lang B. The Structure of Shared Forest in Ambiguous Parsing. In *Proc. of the 27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, 1989.
5. Carpenter B. *The logic of typed feature structures*. Cambridge University Press, Cambridge/New York/Melbourne, 1992.
6. Carroll J. *Practical unification-based parsing of natural language*. Technical report No. 314, University of Cambridge, Computer Laboratory, England. PhD Thesis, 1993.
7. Carroll J. Parsing. In Mitkov R (ed.), *The Oxford Handbook of Computational Linguistics*. Oxford University Press, Oxford, UK, 2003.
8. Clergerie EV. DyALog: a tabular logic programming based environment for NLP. In *Proc. of the 2nd International Workshop on Constraint Solving and Language Processing*, Barcelona, Spain, 2005.
9. Díaz VJ, Alonso MA. Comparing Tabular Parsers for Tree Adjoining Grammars. In *Proc. of Tabulation in Parsing and Deduction (TAPD 2000)*, pages 91–100, Vigo, Spain, 2000.
10. Earley J. An efficient context-free parsing algorithm. *Communications of the ACM,* 13(2):94–102, 1970.
11. Eisner J, Goldlust E, Smith NA. Dyna: A declarative language for implementing dynamic programs. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 218–221, Barcelona, 2004.
12. Gamma E, Helm R, Johnson R, Vlissides J. *Design patterns: Elements of reusable object oriented software*. Addison-Wesley, Reading, Massachusetts, 1995.
13. Gómez-Rodríguez C, Alonso MA, Vilares M. Generating XTAG parsers from algebraic specifications. In *Proc. of TAG+8, the Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms*, Sydney, Australia, 2006.
14. Gómez-Rodríguez C, Alonso MA, Vilares M. On theoretical and practical complexity of TAG parsers. In *FG 2006: The 11th conference on Formal Grammar*. CSLI Online Proceedings. Chapter 5, pp. 61–75, 2006.
15. Gómez-Rodríguez C, Carroll J, Weir D. A Deductive Approach to Dependency Parsing. In *Proc. of The 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL'08:HLT)*, pp. 968–976, Columbus, Ohio, USA, 2008.
16. Gómez-Rodríguez C, Vilares J, Alonso MA. Compiling Declarative Specifications of Parsing Algorithms. In R. Wagner, R. Newell and G. Pernul (eds.), *Database and Expert Systems Applications*, volume 4653 of *Lecture Notes in Computer Science,* pp. 529-538, Springer-Verlag, Berlin-Heidelberg-New York, 2007.
17. Johnson SC. YACC: Yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, USA, 1975.
18. Kasami T. *An efficient recognition and syntax-analysis algorithm for context-free languages*. Scientific Report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA, 1965.
19. Kay M. *Algorithm schemata and data structures in syntactic processing*. Report CSL-80-12, Xerox PARC, Palo Alto, Ca., 1980. Reprinted in: Grosz BJ et al. (Eds.): *Readings in Natural Language Processing*. Morgan Kaufmann, Los Altos, Ca., 1982.
20. Liu YA, Stoller SD. From Datalog rules to efficient programs with time and space guarantees. In *Proc. of the 5th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pp. 172–183, 2003.
21. McAllester D. On the complexity analysis of static analyses. In *Proc. of the 6th International Static Analysis Symposium,* volume 1694 of *Lecture Notes in Computer Science*, pp. 312–329. Springer-Verlag, Berlin, 1999.
22. Nivre J. *Inductive dependency parsing (Text, Speech and Language Technology)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 2006.
23. Rosenkrantz DJ, Lewis II PM. Deterministic Left Corner Parsing. In *Conference Record of 1970 Eleventh Annual Meeting on Switching and Automata Theory*, pages 139–152, 1970.
24. Sarkar A. Practical experiments in parsing using tree adjoining grammars. In *Proc. of TAG+5, the Fifth International Workshop on Tree Adjoining Grammar and Related Formalisms*, Paris, France, 2000.
25. Sampson G. The Susanne corpus, release 3. 1994.
26. Schoorl JJ, Belder S. *Computational linguistics at Delft: A status report*, Report WTM/TT 90–09, 1990.
27. Shieber SM, Schabes Y, Pereira FCN. Principles and implementation of deductive parsing. *Journal of Logic Programming,* 24(1–2):3–36, July-August 1995.
28. Sikkel K. *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.

29. Vijay-Shanker K, Joshi AK. Some Computational Properties of tree Adjoining Grammars. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 82–93, 1985.
30. Viswanadha S. Java Compiler Compiler (JavaCC): The Java Parser Generator. https://javacc.dev.java.net/
31. Younger DH. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2): 189-208, 1967.
32. XTAG Research Group. *A lexicalized tree adjoining grammar for English*. Technical Report IRCS-01-03, IRCS, Univ. of Pennsylvania, 2001.