On TAG Parsing*

Pierre Boullier

INRIA-Rocquencourt
Domaine de Voluceau
B.P. 105
78153 Le Chesnay Cedex, France
E-mail: Pierre.Boullier@inria.fr

Résumé

The subject of tree adjoining grammar parsing inspired many researches but they all failed
to beat the $O(n^6)$ parse time wall. Thus, some present researches in this field try to identify
efficient parser implementations either for the full grammar class, or for linguistically signif-
icant subclasses. This paper addresses both issues and proposes a method which uses range
concatenation grammars as an intermediate implementation formalism. Range concatenation
grammar is a syntactic formalism which is both powerful, in so far as it extends linear context-
free rewriting systems, and efficient, in so far as its sentences can be parsed in polynomial time.
We show two methods by which unrestricted tree adjoining grammars can be transformed into
equivalent range concatenation grammars which can be parsed in $O(n^6)$ time, and, moreover,
if the input tree adjoining grammar has some restricted form, its parse time decreases to $O(n^5)$.

1. Introduction

The subject of tree adjoining grammar (TAG) parsing, or equivalent formalisms such as
linear indexed grammar (LIG), inspired many researches but they all failed to beat the $O(n^6)$
parse time wall.\(^1\) Thus, current researches in this field, can be divided into two parts. A first one
which tries to find parsing methods which can be efficiently implemented, and a second one,
which tries to identify linguistically significant TAG subclasses whose theoretical parse time
is smaller than $O(n^6)$. In this paper, we try to reconcile these two approaches. To reach this
goal, we propose a method which uses range concatenation grammar (RCG) as an intermediate
implementation formalism for TAG parsing.

The notion of RCG is introduced in [Boullier, 1998a]; it is a syntactic formalism which is
a variant of the simple version of literal movement grammar, described in [Groenink, 1997],
and which is also related to the framework of LFP developed by [Rounds, 1988]. This formal-
ism extends context-free grammars (CFGs) and is even more powerful than linear context-free

\(^*\)An extended version of this paper is in [Boullier, 1999a]

\(^1\)Asymptotically faster methods are known, but they all possess large hidden constants. There are even some
arguments (see [Satta, 1994]) against the existence of faster algorithms with small constants.
rewriting systems (LCFRS)\(^2\) \cite{Vijay-ShankerWeirJoshi87}, while staying computationally tractable: its sentences can be parsed in polynomial time. This paper will not address the usage of RCGs as a formalism in which the syntax of natural languages (NLs) can be directly defined. We will rather explore an indirect usage as an intermediate implementation formalism. This usage of RCGs has already been studied in \cite{Boullier98a} for syntactic formalisms such as LIGs or LCFRS. The TAG parsing case has been particularly studied in \cite{Boullier98b}, but the algorithm which transforms a TAG into an equivalent RCG is only proposed for a restricted class of adjunction constraints, and, moreover, the \(\mathcal{O}(n^5)\) parse time is only reached when the original TAG is in a special normal form.

This paper shows two new ways to parse TAGs with RCGs. Both methods accept unrestricted TAGs as inputs, however, the first one directly produces an equivalent RCG which is, in turn, transformed into another RCG, while the second one first dissects the elementary trees of the input TAG before applying a transformation into an object RCG. Both methods produce an equivalent RCG which can be parsed in \(\mathcal{O}(n^5)\) time. Moreover, if the initial TAG is in the restricted form introduced in \cite{SattaSchuler98}, the corresponding RCG has an \(\mathcal{O}(n^5)\) parse time.

\section{Range Concatenation Grammars}

This section introduces the notion of RCG, more details can be found in \cite{Boullier98a}.

A \textit{positive range concatenation grammar} (PRCG) \(G = (N, T, V, P, S)\) is a 5-tuple where \(N\) is a finite set of \textit{predicate names}, \(T\) and \(V\) are finite, disjoint sets of \textit{terminal symbols} and \textit{variable symbols} respectively, \(S \in N\) is the \textit{start predicate name}, and \(P\) is a finite set of clauses

\[
\psi_0 \rightarrow \psi_1 \ldots \psi_m
\]

where \(m \geq 0\) and each of \(\psi_0, \psi_1, \ldots, \psi_m\) is a \textit{predicate} of the form

\[
A(\alpha_1, \ldots, \alpha_i, \ldots, \alpha_p)
\]

where \(p \geq 1\) is its \textit{arity}, \(A \in N\) and each of \(\alpha_i \in (T \cup V)^*\), \(1 \leq i \leq p\), is an \textit{argument}.

Each occurrence of a predicate in the RHS of a clause is a predicate \textit{call}, it is a predicate \textit{definition} if it occurs in its LHS. Clauses which define predicate \(A\) are called \(A\)-clauses. This definition assigns a fixed arity to each predicate name. The arity of the start predicate is one. The \textit{arity} \(k\) of a grammar (we have a \(k\)-PRCG), is the maximum arity of its predicates.

Lower case letters such as \(a, b, c, \ldots\) will denote terminal symbols, while upper case letters such as \(L, R, X, Y, Z\) will denote elements of \(V\).

The language defined by a PRCG is based on the notion of \textit{range}. For a given input string \(w = a_1 \ldots a_n\) a range is a couple \((i, j)\), \(0 \leq i \leq j \leq n\) of integers which denotes the occurrence of some substring \(a_{i+1} \ldots a_j\) in \(w\). The number \(j - i\) is its \textit{size}. We will use several equivalent denotations for ranges: an explicit dotted notation like \(w_1 \cdot w_2 \cdot w_3\) or, if \(w_2\) extends from positions \(i + 1\) through \(j\), a tuple notation \(\langle i .. j \rangle\) or \(\langle i .. j \rangle_w\) when \(w\) is understood or of no importance. For a range \(\langle i .. j \rangle\), \(i\) is its \textit{lower bound} and \(j\) is its \textit{upper bound}. If \(i = j\), we have

\(^2\)In \cite{Boullier99b}, we argue that this extra power can be used in natural language processing.
an empty range. Of course, only consecutive ranges can be concatenated into new ranges. In any PRCG, terminals, variables and arguments in a clause are supposed to be bound to ranges by a substitution mechanism. An instantiated clause is a clause in which variables and arguments are consistently (w.r.t. the concatenation operation) replaced by ranges; its components are instantiated predicates.

For example, \( A((g..h), (i..j), (k..l)) \) \( \rightarrow \) \( B((g+1..h), (i+1..j-1), (k..l-1)) \) is an instantiation of the clause \( A(aX, bYc, Zd) \) \( \rightarrow \) \( B(X, Y, Z) \) if the source text \( a_1 \ldots a_n \) is such that \( a_{g+1} = a, a_{i+1} = b, a_j = c \) and \( a_l = d \). In this case, the variables \( X, Y \) and \( Z \) are bound to \( (g+1..h), (i+1..j-1) \) and \( (k..l-1) \) respectively.

For a given PRCG and an input string \( w \), a derive relation, denoted by \( \Rightarrow_{G,w} \), is defined on strings of instantiated predicates. If an instantiated predicate is the LHS of some instantiated clause, it can be replaced by the RHS of that instantiated clause.

The language of a PRCG \( G = (N, T, V, P, S) \) is the set

\[
\mathcal{L}(G) = \{w \mid S(w \bullet) \Rightarrow_{G,w} \varepsilon\}
\]

An input string \( w = a_1 \ldots a_n \) is a sentence iff the empty string (of instantiated predicates) can be derived from \( S((0..n)) \). Note that the order of predicate calls in the RHS of a clause is of no importance.\(^3\)

The arguments of a given predicate may denote discontinuous or even overlapping ranges. Fundamentally, a predicate name \( A \) defines a notion (property, structure, dependency, ...) between its arguments whose ranges can be arbitrarily scattered over the source text. PRCGs are therefore well suited to describe long distance dependencies. Overlapping ranges arise as a consequence of the non-linearity of the formalism. For example, the same variable (denoting the same range) may occur in different arguments in the RHS of some clause, expressing different views (properties) of the same portion of the source text. However, in this paper, we do not need the full power of RCGs and we will restrict our attention to the simple subclass of PRCGs.

A clause is simple\(^4\) if it is

non-combinatorial: each argument of its RHS predicates consists of a single variable, and

non-erasing and linear: each of its variables appears exactly one time in its LHS and one time in its RHS.

A simple RCG is an RCG in which all its clauses are simple.

As an example, the following simple 3-PRCG defines the three-copy language \( \{www \mid w \in \{a, b\}^*\} \) which is not a CFL and is not even a TAL.

\[
\begin{align*}
S(XYZ) & \rightarrow A(X, Y, Z) \\
A(aX, aY, aZ) & \rightarrow A(X, Y, Z) \\
A(bX, bY, bZ) & \rightarrow A(X, Y, Z) \\
A(\varepsilon, \varepsilon, \varepsilon) & \rightarrow \varepsilon
\end{align*}
\]

\(^3\)In [Boullier, 1998a], we also define negative RCG (NRCG), which allows negative predicate calls. These negative calls define the complement language w.r.t. \( T^* \) of their positive counterpart. A range concatenation grammar (RCG) is either a PRCG or a NRCG.

\(^4\)This is not Groenink’s definition of simple.
A parsing algorithm for RCGs has been presented in [Boullier, 1998a]. For an RCG $G$ and an input string of length $n$, it produces a parse forest in time polynomial with $n$ and linear with $|G|$. The degree of this polynomial is at most the number of free (independent) bounds in any clause. For a simple $k$-RCG, its parse time is at worst $O(|G|^{n^d})$ with $d = \max_{c_j \in P}(k_j + v_j)$, if $c_j$ denotes the $j^{th}$ clause in $P$, $k_j$ is the arity of its LHS predicate and $v_j$ is the number of its variables.

3. First Transformation from Unrestricted TAG to Simple PRCG

The notion of mild context-sensitivity originates in an attempt by [Joshi, 1985] to express the formal power needed to define the syntax of NLs, and the most popular incarnation of mildly context-sensitive formalisms is certainly the TAG formalism. A TAG is a tree rewriting system where trees are composed by means of the operations of adjunction and substitution. Here, we assume that the reader is familiar with TAGs (see [Joshi, 1987] for an introduction).

First, we introduce the notion of decoration strings.

The set of nodes (addresses) in a tree or in a set of trees $\tau$ is denoted by $\mathcal{N}_\tau$. Let $\mathcal{T} = (V_N, T, I, A, S)$ be an input TAG. Every node $\eta \in \mathcal{N}_\tau$ in every elementary (either initial or auxiliary) tree $\tau \in I \cup A$ is decorated as follows.

- If $\eta$ is an adjunction node, it is decorated by two symbols, a left decoration $L_\eta$ and a right decoration $R_\eta$ called its $LR$-variables. These symbols are RCG variables which capture the terminal yields of the complete derived trees that can be adjoined at $\eta$. Its left (resp. right) terminal yield, lays to the left (resp. right) of the foot node and is captured by $L_\eta$ (resp. $R_\eta$).

- If $\eta$ is a substitution node, it is decorated by a single symbol $S_\eta$, called its $S$-variable. This RCG variable captures the terminal yield of the complete derived trees that can be substituted at $\eta$.

- If $\eta$ is a terminal node, it has a single decoration which is either its terminal label or $\varepsilon$.

Afterwards, during a top-down left to right traversal of $\tau$, we collect into a decoration string $\sigma_\tau$ the previous annotations. For an adjunction node, its $L$-variable is collected during its top-down traversal while its $R$-variable is collected during the bottom-up traversal. $S$-variables and terminal decorations, associated with leaves, are gathered during the traversal of these leaf nodes. If $\tau$ is an auxiliary tree, its left decoration string $\sigma^L_\tau$ and its right decoration string $\sigma^R_\tau$ are the parts of $\sigma_\tau$ gathered during the previous traversal, respectively before and after the foot node of $\tau$ (i.e. we have $\sigma^L_\tau = L_{r_\tau} \ldots L_{f_\tau}$ and $\sigma^R_\tau = R_{f_\tau} \ldots R_{r_\tau}$ if $r_\tau$ and $f_\tau$ are the root and foot nodes of $\tau$).

Now, we are ready to describe the TAG to simple PRCG transformation algorithm.

We will generate a simple PRCG $G = (N, T, V, P, S)$ which is equivalent to an initial TAG $\mathcal{T} = (V_N, T, I, A, S)$. We assume that the set $N$ of its predicate names and the set $V$ of its variables are implicitly defined by the clauses in $P$.

In a first phase, for each initial $S$-tree $\alpha$, we initialize $P$ with

$$S(X) \rightarrow \alpha(X)$$
Let $\tau$ be an elementary tree and let $\sigma_\tau$ be its decoration string. Of course, if $\tau$ is an auxiliary tree, $\sigma_\tau$ is cut into its left and right part: $\sigma_\tau = \sigma_\tau^l \sigma_\tau^r$. To each such $\tau$, we associate a simple clause, constructed as follows:

- its LHS is the predicate definition $\alpha(\sigma_\tau)$, if $\tau$ is the initial tree $\alpha$;
- its LHS is the predicate definition $\beta(\sigma_\tau^l, \sigma_\tau^r)$, if $\tau$ is the auxiliary tree $\beta$;
- its RHS is $\psi_1 \ldots \psi_i \ldots \psi_m, 1 \leq i \leq m, m = |N_\tau|$ with
  - $\psi_i = [adj(\eta_i)](L_{\eta_i}, R_{\eta_i})$, if $\eta_i$ is an adjunction node in $N_\tau$,
  - $\psi_i = [sbst(\eta_i)](S_{\eta_i})$, if $\eta_i$ is a substitution node in $N_\tau$, and
  - $\psi_i = \varepsilon$, if $\eta_i$ is a terminal node in $N_\tau$.

The denotations $[adj(\eta_i)]$ and $[sbst(\eta_i)]$ are predicate names which respectively symbolize the adjunction or substitution operations that can be performed at node $\eta_i$.

For each nonterminal node $\eta \in N_{\mathcal{I} \cup A}$, we define these predicates by the following clauses.

- If $\eta$ is an adjunction node, and if $adj$ is the adjunction constraint function, whose value is the set of auxiliary trees that can be adjoined at $\eta$ (we write $nil \in adj(\eta)$, for an optional adjunction), then, for every $\tau \in adj(\eta)$ we produce the clause
  \[
  [adj(\eta)](L, R) \rightarrow \tau(L, R)
  \]
  if $\tau \in \mathcal{A}$ or
  \[
  [adj(\eta)](\varepsilon, \varepsilon) \rightarrow \varepsilon
  \]
  if $\tau = nil$.

- If $\eta$ is a substitution node, and if $sbst$ is the substitution constraint function, whose value is the set of initial trees that can be substituted at $\eta$ (we write $nil \in sbst(\eta)$, for an optional substitution), then, for every $\tau \in sbst(\eta)$ we produce the clause
  \[
  [sbst(\eta)](S) \rightarrow \tau(S)
  \]
  if $\tau \in \mathcal{I}$ or
  \[
  [sbst(\eta)](\varepsilon) \rightarrow \varepsilon
  \]
  if $\tau = nil$.

Since these clauses are all simple and positive, and since predicates are all at most binaries, $G$ is a simple 2-PRCG.\(^5\)

Let us now examine the complexity of the parsing algorithm for $G$. Applying to our simple 2-PRCG the general result on parsing with simple $k$-RCG, we get an $O(n^{v+2})$ parse time. Expressed in terms of the original TAG, and since there are two $LR$-variables per adjunction

\(^5\)In this paper, we will not address the correctness of the previous algorithm and we assume that it generates a PRCG which is equivalent to the original TAG.
node, in the worst case, we have an $O(n^{2p+2})$ parse time for unrestricted TAGs, if $p$ is the maximum number of adjunction nodes in an elementary tree.

The idea is now to see whether $G$ can be transformed into an equivalent RCG $G'$ with a better parse time, and in particular whether we can reached the classical $O(n^6)$ bound. The purpose of what follows is to transform each previously generated clause into a sequence of equivalent clauses in such a way that the number of their variables (and thus the number of their free bounds) is as small as possible.

If we consider the decoration string $\sigma_\tau$ associated with any elementary tree $\tau$, it is not difficult to figure out that $\sigma_\tau$ is a well parenthesized (Dyck) string where, on the one hand, the couples of parentheses are the $L$ and $R$-variables associated with its adjunction nodes, and, on the other hand, the basic vocabulary is formed both by $T$, the set of terminal symbols, and by the set of its $S$-variables.

Fundamentally, a Dyck language is recursively defined from initial strings in its basic vocabulary either by concatenation of two Dyck languages or by wrapping a Dyck language into a couple of parentheses. Conversely, each Dyck string can be recursively and unambiguously decomposed (parsed) either into concatenation of two Dyck strings or into a wrapped Dyck string. In our case, at each step of such decomposition of a Dyck (decoration) string $\sigma$, we can associate an RCG clause which exhibits, either its cutting into a wrapped prefix part $\sigma_1$ and a suffix part $\sigma_2$, or its unwrapping. However, this process is slightly complicated by the fact that $\sigma$, in the auxiliary tree case, is itself decomposed into two arguments, a left argument $\sigma^l$ and a right argument $\sigma^r$. In [Boullier, 1999a], we show that this decomposition produces, for each original clause, a sequence of equivalent clauses which, in the worst case, define binary predicates with four variables. Therefore, we have a method which parses the language defined by an unrestricted TAG in worst case time $O(n^6)$.

4. Second Transformation from Unrestricted TAG to Simple PRCG

The idea of this second algorithm is to dissect the elementary trees of a TAG in such a way that each excised subtree directly generates at worst an $O(n^6)$ time parsable clause.

The same symbol $\alpha$ or $\beta$ will be used to denote both any elementary tree and the root node of that elementary tree. If $\eta$ is not a leaf node in some elementary tree, its $i^{th}$ daughter, from left to right, is denoted $\eta_i$. A node on a spine is termed as spinal node. If the root of a (sub)tree is a spinal node, we have a spinal tree. Thus, a spinal tree is a subtree of an auxiliary tree rooted on its spine. A tree rooted at node $\eta$ is denoted either by $\eta$ if it is a spinal tree or by $\eta^o$ if it is a non spinal tree. The over hanging circle depicts its root node while the underlying triangles depict its subtrees. For a given node $\eta$, we define a full open tree as the list of its daughter trees. This full open tree is noted $\eta_f$ or $\eta$ according as $\eta$ is a spinal or a non spinal node. An open tree is a list of consecutive daughter trees $\{\eta_i, \eta_{i+1}, \ldots, \eta_j\}$. In the sequel we will handle two kinds of open trees $\eta$ and $\eta$ where $\eta$ denotes the left daughters whose rank is less or equal than $i$ and $\eta$ denotes the right daughters whose rank is greater or equal than $i$. By contrast with the term open, a “normal” tree will be sometimes called a close tree.

---

6Their number is linear in the number of adjunction nodes in the original elementary tree.
7This most costly case corresponds to an adjunction at a spinal node.
8Note that the initial trees are also denoted by $\eta^o$, and the auxiliary trees by $\eta^o$. 

---

Pierre Boullier
For a given TAG $\mathcal{T} = (V_N, T, \mathcal{I}, \mathcal{A}, S)$, we will build an equivalent simple 2-PRCG $G = (N, T, V, P, S)$. Except for $S$, the predicates names of $N$ are the previously defined tree denotations. The *spinal tree* predicates are binary while the *non spinal tree* predicates are unary. The set of clauses $P$ is built, following the transformation rules listed below, each clause introduces its own variables.

In a first phase, for each initial $S$-tree $\alpha$, we initialize $P$ with

$$S(X) \rightarrow \dot{\alpha}_\Delta (X) \quad (1)$$

Afterwards, the elementary trees of $\mathcal{I} \cup \mathcal{A}$ are processed in turn. Each elementary tree is cut into smaller pieces, starting from the root. Intuitively, at each step, a (sub)tree is cut into two parts, its root and its full open tree. The type of processing for the root node depends whether adjunctions are allowed or not. While the processing of a full open tree consists of a partitioning into its constituent parts which are either (smaller) open trees or close trees. Of course, this processing differs whether spinal or non spinal open trees are considered. We iterate until the leaves are reached. If we reach a substitution node, we apply the possible substitutions.

We first consider the transformation of non spinal close trees rooted at node $\eta$.

If $\dot{\eta}$ is a non trivial tree, we have

$$\eta \quad \Rightarrow \quad \left\{ \begin{array} { l l l } { \beta \in \text{adj}(\eta), } & { \dot{\eta}_\Delta (L \, X \, R) } & { \rightarrow } \quad \dot{\beta}_{\text{sbst}} (L, R) \, \eta_\Delta (X) \quad (a) \\ { \text{nil} \in \text{adj}(\eta), } & { \dot{\eta}_\Delta (X) } & { \rightarrow } \quad \eta_\Delta (X) \quad (b) \end{array} \right.$$  

(2)

If $\dot{\eta}$ is a leaf tree, depending of its label, we have

for an empty leaf (i.e. $\varepsilon = \text{lab}(\eta)$)

$$\eta \varepsilon \quad \Rightarrow \quad \dot{\eta}_\Delta (\varepsilon) \rightarrow \varepsilon \quad (3)$$

for a terminal leaf $a$ (i.e. $a = \text{lab}(\eta), a \in T$)

$$\eta \, a \quad \Rightarrow \quad \dot{\eta}_\Delta (a) \rightarrow \varepsilon \quad (4)$$

for a substitution node (i.e. $A = \text{lab}(\eta), A \in V_N$)

$$\eta \, A \quad \Rightarrow \quad \left\{ \begin{array} { l l l } { \alpha \in \text{sbst}(\eta), } & { \dot{\eta}_\Delta (X) } & { \rightarrow } \quad \dot{\alpha}_\Delta (X) \quad (a) \\ { \text{nil} \in \text{sbst}(\eta), } & { \dot{\eta}_\Delta (\varepsilon) } & { \rightarrow } \quad \varepsilon \quad (b) \end{array} \right.$$  

(5)
Now, we consider the transformation of non spinal open trees rooted at \( \eta \). The components of such trees are processed from left to right in the case \( \eta \) or from right to left in the case \( \eta' \).

For an open tree of the form \( \eta \), we know that the left daughters of \( \eta \) whose rank is less than \( i \) must not be considered, thus such an open tree is processed by extracting its \( i \)th daughter close tree and by iteratively processing \( \eta' \), until completion. Of course, this completion is reached when, after the extraction of the \( i \)th daughter, the resulting open tree is empty.

For a right to left processing, we have

\[
\begin{align*}
\eta &\quad \Rightarrow \quad \begin{cases} 
Y \neq \emptyset, & \eta \ (X \ Y) \quad \rightarrow \quad \eta_{i+1} \ (Y) \quad (a) \\
Y = \emptyset, & \eta \ (X) \quad \rightarrow \quad \eta_{i} \ (X) \quad (b)
\end{cases}
\end{align*}
\]

(6)

Now, we consider the transformation of spinal trees rooted at \( \eta \).

For each spinal close tree such as \( \eta \), we must handle both a possible set of adjunctions at \( \eta \), and the processing of the full spinal open subtree \( \eta' \). Thus, we have the transformation rule

\[
\begin{align*}
\eta &\quad \Rightarrow \quad \begin{cases} 
\beta \in \text{adj}(\eta), & \eta (LX, Y R) \quad \rightarrow \quad \beta (L, R) \ \eta \ (X, Y) \quad (a) \\
nil \in \text{adj}(\eta), & \eta (X, Y) \quad \rightarrow \quad \eta \ (X, Y) \quad (b)
\end{cases}
\end{align*}
\]

(8)

For each spinal open tree \( \eta' \), we have, if \( \eta' \) is the spinal daughter node

\[
\begin{align*}
\eta &\quad \Rightarrow \quad \eta \ (LX, Y R) \quad \rightarrow \quad \eta_1 \ (L) \ \eta' \ (X, Y) \ \eta \ (R)
\end{align*}
\]

(9)

In that case, the number of free bounds in the corresponding clause is six. We can remark that, on the one hand, the left and right open trees \( \eta \) and \( \eta' \) are independent and may thus be computed one after the other and, on the other hand, that any of them, or both, can be empty. Thus, equivalently, if \( \eta_r \) denotes an intermediate predicate name, we have the following transformation rules
5. Conclusion

In this paper, we proposed two methods to implement a TAG parser in $O(n^6)$ time. These algorithms both use RCGs, a powerful high level syntactic description formalism, as an intermediate object language. Since [Boullier, 1998b], we know that RCGs can be used to implement parsers for TAGs. However, the $O(n^6)$ bound was only reached with a restricted form of adjunction constraints and when the initial TAG was in some normal form. In particular, this normal form assumes that elementary trees are in a binary branching form, from in which the original structure has disappeared. At the contrary, the algorithms presented here, work for completely unrestricted TAGs though their corresponding parsers still work in $O(n^6)$ time at worst.

---

9As for the first algorithm, we leave aside the proof of the correctness of the above algorithm.
In [Satta and Schuler, 1998], the authors introduced a linguistically significant restricted form of TAGs that can be parsed in $O(n^5)$, in [Boullier, 1999a], we have shown that their subclass can be transformed into an equivalent RCG that can also be parsed in $O(n^5)$ time, whether we start from their inference rules, or directly from the elementary trees representation.

The usage of RCGs as an intermediate structure may result in several advantages. First, since the RCG formalism is simple, the transformations proposed in this paper give another view of the (rather complicated) adjunction mechanism and may help to understand when and why it is costly to implement. Second, since RCGs can be efficiently implemented, we are convinced that these methods are good candidates for practical TAG implementations.

Références


