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 $\mathcal{O}(n^6)$ 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 significant 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 $\mathcal{O}(n^6)$ time, and, moreover, if the input tree adjoining grammar has some restricted form, its parse time decreases to $\mathcal{O}(n^5)$.

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 $\mathcal{O}(n^6)$ 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 $\mathcal{O}(n^6)$. 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 formalism 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] [Vijay-Shanker, Weir, and Joshi, 1987], 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 [Boullier, 1998a] for syntactic formalisms such as LIGs or LCFRS. The TAG parsing case has been particularly studied in [Boullier, 1998b], but the algorithm which transforms a TAG into an equivalent RCG is only proposed for a restricted class of adjunction constraints, and, moreover, the $\mathcal{O}(n^6)$ 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 $\mathcal{O}(n^6)$ time. Moreover, if the initial TAG is in the restricted form introduced in [Satta and Schuler, 1998], the corresponding RCG has an $\mathcal{O}(n^5)$ parse time.

2. Range Concatenation Grammars

This section introduces the notion of RCG, more details can be found in [Boullier, 1998a].

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

$$\psi_0 \rightarrow \psi_1 \dots \psi_m$$

where $m \geq 0$ and each of $\psi_0, \psi_1, \dots, \psi_m$ is a *predicate* of the form

$$A(\alpha_1, \dots, \alpha_i, \dots, \alpha_p)$$

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

Each occurrence of a predicate in the RHS of a clause is a predicate *call*, it is a predicate *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 *arity* k of a grammar (we have a k-PRCG), is the maximum arity of its predicates.

Lower case letters such as $a, b, c, \dots$ 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 *range*. For a given input string $w = a_1 \dots a_n$ a range is a couple (i, j), $0 \leq i \leq j \leq n$ of integers which denotes the occurrence of some substring $a_{i+1} \dots a_j$ in w. The number $j - i$ is its *size*. We will use several equivalent denotations for ranges: an explicit dotted notation like $w_1 \bullet w_2 \bullet w_3$ or, if w_2 extends from positions $i + 1$ through j, a tuple notation $\langle i..j \rangle_w$, or $\langle i..j \rangle$ when w is understood or of no importance. For a range $\langle i..j \rangle$, i is its *lower bound* and j is its *upper bound*. If $i = j$, we have

[2] In [Boullier, 1999b], 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, $A(\langle g..h\rangle, \langle i..j\rangle, \langle k..l\rangle) \rightarrow B(\langle g{+}1..h\rangle, \langle i{+}1..j{-}1\rangle, \langle k..l{-}1\rangle)$ 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 $\langle g{+}1..h\rangle$, $\langle i{+}1..j{-}1\rangle$ and $\langle k..l{-}1\rangle$ respectively.

For a given PRCG and an input string w, a *derive* relation, denoted by $\underset{G,w}{\Rightarrow}$, 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

$$\mathcal{L}(G) \quad = \quad \{w \mid S(\bullet w \bullet) \overset{+}{\underset{G,w}{\Rightarrow}} \varepsilon\}$$

An input string $w = a_1 \ldots a_n$ is a sentence iff the empty string (of instantiated predicates) can be derived from $S(\langle 0..n\rangle)$. 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.

$$\begin{array}{lcl} S(XYZ) & \rightarrow & A(X, Y, Z) \\ A(aX, aY, aZ) & \rightarrow & A(X, Y, Z) \\ A(bX, bY, bZ) & \rightarrow & A(X, Y, Z) \\ A(\varepsilon, \varepsilon, \varepsilon) & \rightarrow & \varepsilon \end{array}$$

[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 $\mathcal{O}(|G|n^d)$ 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 τ is denoted by $\mathcal{N}_\tau$. Let $\mathcal{T} = (V_N, T, \mathcal{I}, \mathcal{A}, S)$ be an input TAG. Every node $\eta \in \mathcal{N}_\tau$ in every elementary (either initial or auxiliary) tree $\tau \in \mathcal{I} \cup \mathcal{A}$ is decorated as follows.

- If η is an adjunction node, it is decorated by two symbols, a left decoration L_η and a right decoration R_η called its LR-*variables*. These symbols are RCG variables which capture the terminal yields of the complete derived trees that can be adjoined at η. Its left (resp. right) terminal yield, lays to the left (resp. right) of the foot node and is captured by L_η (resp. R_η).
- If η is an substitution node, it is decorated by a single symbol S_η, called its S-*variable*. This RCG variable captures the terminal yield of the complete derived trees that can be substituted at η.
- If η is a terminal node, it has a single decoration which is either its terminal label or ε.

Afterwards, during a top-down left to right traversal of τ, we collect into a *decoration string* σ_τ 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 τ is an auxiliary tree, its *left* decoration string σ^l_τ and its *right* decoration string σ^r_τ are the parts of σ_τ gathered during the previous traversal, respectively before and after the foot node of τ (i.e. we have $\sigma^l_\tau = L_{r_\tau} \dots L_{f_\tau}$ and $\sigma^r_\tau = R_{f_\tau} \dots R_{r_\tau}$ if r_τ and f_τ are the root and foot nodes of τ).

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 $\mathcal{T} = (V_N, T, \mathcal{I}, \mathcal{A}, S)$. 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 α, we initialize P with

$$S(X) \quad \rightarrow \quad \alpha(X)$$

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

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

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 η_i.

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

- If η 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 η (we write $nil \in adj(\eta)$, for an optional adjunction), then, for every $\tau \in adj(\eta)$ we produce the clause

$$[adj(\eta)](L, R) \quad \rightarrow \quad \tau(L, R)$$

  if $\tau \in \mathcal{A}$ or

$$[adj(\eta)](\varepsilon, \varepsilon) \quad \rightarrow \quad \varepsilon$$

  if $\tau = nil$.

- If η 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 η (we write $nil \in sbst(\eta)$, for an optional substitution), then, for every $\tau \in sbst(\eta)$ we produce the clause

$$[sbst(\eta)](S) \quad \rightarrow \quad \tau(S)$$

  if $\tau \in \mathcal{I}$ or

$$[sbst(\eta)](\varepsilon) \quad \rightarrow \quad \varepsilon$$

  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 $\mathcal{O}(n^{v+2})$ 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 $\mathcal{O}(n^{2p+2})$ 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 $\mathcal{O}(n^6)$ 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 σ_τ associated with any elementary tree τ, it is not difficult to figure out that σ_τ 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 σ, we can associate an RCG clause which exhibits, either its cutting into a wrapped prefix part σ_1 and a suffix part σ_2, or its unwrapping. However, this process is slightly complicated by the fact that σ, in the auxiliary tree case, is itself decomposed into two arguments, a left argument σ^l and a right argument σ^r. In [Boullier, 1999a], we show that this decomposition produces, for each original clause, a sequence of equivalent clauses[6] which, in the worst case, define binary predicates with four variables.[7] Therefore, we have a method which parses the language defined by an unrestricted TAG in worst case time $\mathcal{O}(n^6)$.

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 $\mathcal{O}(n^6)$ time parsable clause.

The same symbol α or β will be used to denote both any elementary tree and the root node of that elementary tree. If η 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 η is denoted either by $\overset{\circ}{\underset{\lhd\rhd}{\eta}}$ if it is a spinal tree or by $\overset{\circ}{\underset{\triangle}{\eta}}$ if it is a non spinal tree.[8] The over hanging circle depicts its root node while the underlying triangles depict its subtrees. For a given node η, we define a *full open* tree as the list of its daughter trees. This full open tree is noted $\underset{\lhd\rhd}{\eta}$ or $\underset{\triangle}{\eta}$ according as η 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 $\underset{\lhd i}{\eta}$ and $\underset{i\rhd}{\eta}$ where $\underset{\lhd i}{\eta}$ denotes the left daughters whose rank is less or equal than i and $\underset{i\rhd}{\eta}$ 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.

[6] Their number is linear in the number of adjunction nodes in the original elementary tree.

[7] This most costly case corresponds to an adjunction at a spinal node.

[8] Note that the α initial trees are also denoted by $\overset{\circ}{\underset{\triangle}{\alpha}}$, and the β auxiliary trees by $\overset{\circ}{\underset{\lhd\rhd}{\beta}}$.

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 α, we initialize P with

$$S(X) \rightarrow \underset{\triangle}{\overset{\circ}{\alpha}}(X) \tag{1}$$

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 η.

If $\underset{\triangle}{\overset{\circ}{\eta}}$ is a non trivial tree, we have

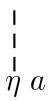
$$\Longrightarrow \begin{cases} \beta \in adj(\eta), & \underset{\triangle}{\overset{\circ}{\eta}}(L\ X\ R) \rightarrow \underset{\triangleleft\triangleright}{\overset{\circ}{\beta}}(L,R)\ \underset{1\triangleright}{\eta}(X) & (a) \\ nil \in adj(\eta), & \underset{\triangle}{\overset{\circ}{\eta}}(X) \rightarrow \underset{1\triangleright}{\eta}(X) & (b) \end{cases} \tag{2}$$

If $\underset{\triangle}{\overset{\circ}{\eta}}$ is a leaf tree, depending of its label, we have

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

$$\Longrightarrow \underset{\triangle}{\overset{\circ}{\eta}}(\varepsilon) \rightarrow \varepsilon \tag{3}$$

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



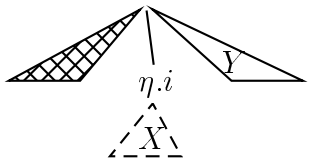
$$\Longrightarrow \underset{\triangle}{\overset{\circ}{\eta}}(a) \rightarrow \varepsilon \tag{4}$$

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

$$\Longrightarrow \begin{cases} \alpha \in sbst(\eta), & \underset{\triangle}{\overset{\circ}{\eta}}(X) \rightarrow \underset{\triangle}{\overset{\circ}{\alpha}}(X) & (a) \\ nil \in sbst(\eta), & \underset{\triangle}{\overset{\circ}{\eta}}(\varepsilon) \rightarrow \varepsilon & (b) \end{cases} \tag{5}$$

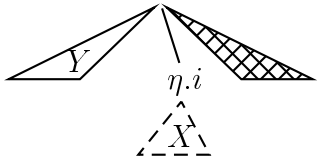
Now, we consider the transformation of non spinal open trees rooted at η. The components of such trees are processed from left to right in the case $\underset{i\triangleright}{\eta}$ or from right to left in the case $\underset{\triangleleft i}{\eta}$.

For an open tree of the form $\underset{i\triangleright}{\eta}$, we know that the left daughters of η 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 $\underset{i+1\triangleright}{\eta}$, until completion. Of course, this completion is reached when, after the extraction of the i^{th} daughter, the resulting open tree is empty.



$$\Longrightarrow \quad \begin{cases} Y \neq \emptyset, & \underset{i\triangleright}{\eta}\,(X\ Y) & \rightarrow & \overset{\circ}{\underset{\triangle}{\eta.i}}\,(X) & \underset{i+1\triangleright}{\eta}\,(Y) & (a) \\ Y = \emptyset, & \underset{i\triangleright}{\eta}\,(X) & \rightarrow & \overset{\circ}{\underset{\triangle}{\eta.i}}\,(X) & & (b) \end{cases} \tag{6}$$

For a right to left processing, we have



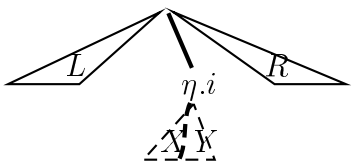
$$\Longrightarrow \quad \begin{cases} i > 1, & \underset{\triangleleft i}{\eta}\,(Y\ X) & \rightarrow & \underset{\triangleleft i\text{-}1}{\eta}\,(Y) & \overset{\circ}{\underset{\triangle}{\eta.i}}\,(X) & (a) \\ i = 1, & \underset{\triangleleft 1}{\eta}\,(X) & \rightarrow & \overset{\circ}{\underset{\triangle}{\eta.1}}\,(X) & & (b) \end{cases} \tag{7}$$

Now, we consider the transformation of spinal trees rooted at η.

For each spinal close tree such as $\overset{\circ}{\underset{\triangleleft\triangleright}{\eta}}$, we must handle both a possible set of adjunctions at η, and the processing of the full spinal open subtree $\underset{\triangleleft\triangleright}{\eta}$. Thus, we have the transformation rule

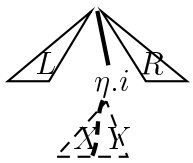
$$\Longrightarrow \quad \begin{cases} \beta \in adj(\eta), & \overset{\circ}{\underset{\triangleleft\triangleright}{\eta}}\,(LX, YR) & \rightarrow & \overset{\circ}{\underset{\triangleleft\triangleright}{\beta}}\,(L, R) & \underset{\triangleleft\triangleright}{\eta}\,(X, Y) & (a) \\ nil \in adj(\eta), & \overset{\circ}{\underset{\triangleleft\triangleright}{\eta}}\,(X, Y) & \rightarrow & \underset{\triangleleft\triangleright}{\eta}\,(X, Y) & & (b) \end{cases} \tag{8}$$

For each spinal open tree $\underset{\triangleleft\triangleright}{\eta}$, we have, if $\eta.i$ is the spinal daughter node



$$\Longrightarrow \quad \underset{\triangleleft\triangleright}{\eta}\,(LX, YR) \ \rightarrow \ \underset{\triangleleft i\text{-}1}{\eta}\,(L)\ \overset{\circ}{\underset{\triangleleft\triangleright}{\eta.i}}\,(X, Y)\ \underset{i+1\triangleright}{\eta}\,(R)$$

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 $\underset{\triangleleft i\text{-}1}{\eta}$ and $\underset{i+1\triangleright}{\eta}$ 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 η_r denotes an intermediate predicate name, we have the following transformation rules



$$\Longrightarrow \begin{cases} L \neq \emptyset, R \neq \emptyset, \begin{cases} \underset{\lhd\rhd}{\eta}\,(LX,Y) \rightarrow \underset{\lhd i\text{-}1}{\eta}\,(L)\;\eta_r(X,Y) & (a) \\ \eta_r(X,YR) \rightarrow \underset{\lhd\rhd}{\overset{\circ}{\eta.i}}\,(X,Y)\;\underset{i+1\rhd}{\eta}\,(R) & (b) \end{cases} \\ L = \emptyset, R \neq \emptyset, \quad \underset{\lhd\rhd}{\eta}\,(X,YR) \rightarrow \underset{\lhd\rhd}{\overset{\circ}{\eta.i}}\,(X,Y)\;\underset{i+1\rhd}{\eta}\,(R) & (c) \\ L \neq \emptyset, R = \emptyset, \quad \underset{\lhd\rhd}{\eta}\,(LX,Y) \rightarrow \underset{\lhd i\text{-}1}{\eta}\,(L)\;\underset{\lhd\rhd}{\overset{\circ}{\eta.i}}\,(X,Y) & (d) \\ L = \emptyset, R = \emptyset, \quad \underset{\lhd\rhd}{\eta}\,(X,Y) \rightarrow \underset{\lhd\rhd}{\overset{\circ}{\eta.i}}\,(X,Y) & (e) \end{cases} \tag{9}$$

where the number of free bounds is now less or equal to five.

And last, for a foot node η, we have

$$\Longrightarrow \begin{cases} \beta \in adj(\eta), \quad \underset{\lhd\rhd}{\overset{\circ}{\eta}}\,(L,R) \rightarrow \underset{\lhd\rhd}{\overset{\circ}{\beta}}\,(L,R) & (a) \\ nil \in adj(\eta), \quad \underset{\lhd\rhd}{\overset{\circ}{\eta}}\,(\varepsilon,\varepsilon) \rightarrow \varepsilon & (b) \end{cases} \tag{10}$$

By inspection of the generated clauses, we can verify that we have a simple 2-PRCG. The degree of the polynomial time complexity of this grammar is the maximum number of free bounds in each clause. We see that the maximum number of free bounds is six and is only reached one time in the clause resulting of transformation (8a). Transformations (9a–d) cost five (recall that they all arise as an optimization of a six free bound clause), while all the others contribute to a number less or equal to four. This transformation explicitly ranks by cost the operations used in TAG parsing. In particular, we confirm that the most costly operation is an adjunction at a spinal node.[9]

We have shown that each TAG can be translated into an equivalent simple 2-PRCG which can be parsed, at worst, in $\mathcal{O}(n^6)$ time.

To account for the dependency from the input grammar size, we define $|\mathcal{T}| = \sum_{\eta \in \mathcal{N}_{\mathcal{I}\cup\mathcal{A}}}(1 + |adj(\eta)| + |sbst(\eta)|)$. We can easily see that the number of generated clauses is proportional to $|\mathcal{T}|$. Since the parse time of an RCG is linear in the size of its input grammar, we finally get an $\mathcal{O}(|\mathcal{T}|\, n^6)$ parse time for G.

5. Conclusion

In this paper, we proposed two methods to implement a TAG parser in $\mathcal{O}(n^6)$ 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 $\mathcal{O}(n^6)$ 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, form 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 $\mathcal{O}(n^6)$ time at worst.

[9] As 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 $\mathcal{O}(n^5)$, in [Boullier, 1999a], we have shown that their subclass can be transformed into an equivalent RCG that can also be parsed in $\mathcal{O}(n^5)$ 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

[Boullier, 1998a] Boullier P. (1998). Proposal for a Natural Language Processing Syntactic Backbone. In *Research Report No 3342* at `http://www.inria.fr/RRRT/RR-3342.html`, INRIA-Rocquencourt, France, Jan. 1998, 41 pages.

[Boullier, 1998b] Boullier P. (1998). A Generalization of Mildly Context-Sensitive Formalisms. In *Proceedings of the Fourth International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+4)*, University of Pennsylvania, Philadelphia, PA, 1–3 August, pages 17–20.

[Boullier, 1999a] Boullier P. (1999). On TAG and Multicomponent TAG parsing. In *Research Report No 3668* at `http://www.inria.fr/RRRT/RR-3668.html`, INRIA-Rocquencourt, France, Apr. 1999, 39 pages.

[Boullier, 1999b] Boullier P. (1999). Chinese Numbers, MIX, Scrambling, and Range Concatenation Grammars In *Proceedings of the 9th Conference of the Europwean Chapter of the Association for Computational Linguistics (EACL'99)*, Bergen, Norway, June 8–12.

[Groenink, 1997] Groenink A. (1997). SURFACE WITHOUT STRUCTURE Word order and tractability issues in natural language analysis. PhD thesis, Utrecht University, The Netherlands, Nov. 1977, 250 pages.

[Joshi, 1985] Joshi A. (1985). How much context-sensitivity is necessary for characterizing structural descriptions — Tree Adjoining Grammars. In *Natural Language Processing — Theoritical, Computational and Psychological Perspective*, D. Dowty, L. Karttunen, and A. Zwicky, editors, Cambridge University Press, New-York, NY.

[Joshi, 1987] Joshi A. (1987). An Introduction to Tree Adjoining Grammars. In *Mathematics of Language*, Manaster-Ramer, A., editors, John Benjamins, Amsterdam, pages 87–114.

[Rounds, 1988] Rounds W. (1988). LFP: A Logic for Linguistic Descriptions and an Analysis of its Complexity. In *ACL Computational Linguistics*, Vol. 14, No. 4, pages 1–9.

[Satta, 1994] Satta G. (1994). Tree adjoining grammars parsing and boolean matrix multiplication. In *Computational Linguistics*, 20(2), pages 173–192.

[Satta and Schuler, 1998] Satta G. and Schuler W. (1998). Restrictions on Tree Adjoining Languages. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics (COLING-ACL'98)*, Université de Montréal, Montréal, Québec, Canada, 10–14 August, vol. II, pages 1176–1182.

[Vijay-Shanker, Weir, and Joshi, 1987] Vijay-Shanker K., Weir D. and Joshi A. (1987). Characterizing Structural Descriptions Produced by Various Grammatical Formalisms. In *Proceedings of the 25th Meeting of the Association for Computational Linguistics (ACL'87)*, Stanford University, CA, pages 104–111.