BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 58, No. 3, 2010
DOI: 10.2478/v10175-010-0036-0

TuLiPA – Parsing extensions of TAG with range concatenation grammars

L. KALLMEYER[1*], W. MAIER[1], Y. PARMENTIER[2] and J. DELLERT[1]

[1] SFB 833, Universität Tübingen, Nauklerstr. 35, D-72 074 Tübingen, Germany
[2] LIFO, Université d'Orléans, Bât. 3IA, Rue Léonard De Vinci – BP 6759 – F-45 067 Orléans Cedex 2, France

Abstract. In this paper we present a parsing framework for extensions of Tree Adjoining Grammar (TAG) called TuLiPA (Tübingen Linguistic Parsing Architecture). In particular, besides TAG, the parser can process Tree-Tuple MCTAG with Shared Nodes (TT-MCTAG), a TAG-extension which has been proposed to deal with scrambling in free word order languages such as German. The central strategy of the parser is such that the incoming TT-MCTAG (or TAG) is transformed into an equivalent Range Concatenation Grammar (RCG) which, in turn, is then used for parsing. The RCG parser is an incremental Earley-style chart parser. In addition to the syntactic anlysis, TuLiPA computes also an underspecified semantic analysis for grammars that are equipped with semantic representations.

Key words: tree-adjoining grammar, parsing, range concatenation grammar.

1. Introduction

The starting point of the work presented here is the aim to implement a parser for a German TAG-based grammar that computes syntax and semantic representations. As a grammar formalism for German we choose a multicomponent extension of TAG called Tree Tuple Multicomponent TAG with Shared Nodes (TT-MCTAG) which has been first introduced by Lichte [1], in order to model free-word order phenomena. Instead of implementing a specific TT-MCTAG parser we follow a more general approach by using Range Concatenation Grammar (RCG) as a pivot formalism. More specifically, for TAG and TT-MCTAG we use so-called *simple* RCG. The TT-MCTAG (or TAG) is transformed into a strongly equivalent RCG that is then used for parsing. The motivation for the passage via RCG is that the RCG directly represents the set of derivation trees of the original grammar. Consequently, it abstracts away from traversals of elementary trees combined via substitutions or adjunctions and the output of the RCG parser can be directly interpreted as the TT-MCTAG (or TAG) derivation forest.

We have implemented the conversion into RCG, the RCG parser and the retrieval of the corresponding TT-MCTAG analyses. The parsing architecture comes with graphical input and output interfaces, and an XML export of the result of parsing, for integration within an NLP application. TuLiPA is freely available under the terms of the GNU General Public License[1].

In this paper, we present this parsing architecture focussing on the following aspects: first, we introduce the TT-MCTAG formalism (Sec. 2). Then, we present successively the RCG formalism and the conversion of TT-MCTAG into simple RCG (Sec. 3). Section 4 explains the parsing algorithm(s) we use for the specific RCGs we obtain from TAG and TT-MCTAG. In Sec. 5, we present the implementation of the RCG-based parsing architecture. Finally, we conclude presenting perspectives for future work.

2. TT-MCTAG

2.1. Tree adjoining grammars. Tree Adjoining Grammar [2] is a formalism based on tree rewriting. We briefly summarize here the relevant definitions and refer the reader to [2] for a more complete introduction.

Definition 1 (Tree Adjoining Grammar). A Tree Adjoining Grammar (TAG) is a tuple $G = (V_N, V_T, S, I, A)$ where

- V_N and V_T are disjoint alphabets of non-terminal and terminal symbols, respectively,
- $S \in V_N$ is the start symbol,
- and I and A are finite sets of initial and auxiliary trees, respectively.

Trees in $I \cup A$ are called elementary trees. The internal nodes in the elementary trees are labeled with non-terminal symbols, the leaves with non-terminal or terminal symbols. As a special property, each auxiliary tree β has exactly one of its leaf nodes having the same label as the root. This leaf node is called foot node and marked with a *. The foot node is used in the tree rewriting operation called adjunction introduced below. Leaves with non-terminal labels that are not foot nodes are called substitution nodes.

On top of this tuple-based definition of TAG, one can define the Obligatory-Adjunction function f_{OA} as follows:

$$f_{OA} : \{n \mid n \text{ a node in some } \gamma \in I \cup A\} \rightarrow \{0,1\}$$

For a given node n, the function f_{OA} specifies whether adjunction is obligatory (value 1) or not (value 0).

*e-mail: lk@sfs.uni-tuebingen.de
[1] See http://sourcesup.cru.fr/tulipa/

In a TAG, larger trees can be derived from the elementary trees by subsequent applications of the operations substitution and adjunction. The substitution operation replaces a substitution node η with an initial tree having a root node with the same label as η. The adjunction operation replaces an internal node η in a previously derived tree γ with an auxiliary tree β having root and foot nodes with the same label as η. The subtree of γ rooted at η is then placed below the foot node of β. Only internal nodes can allow for adjunction, in other words, adjunction at leaves is not possible. See Fig. 1 for an example of a tree derivation.

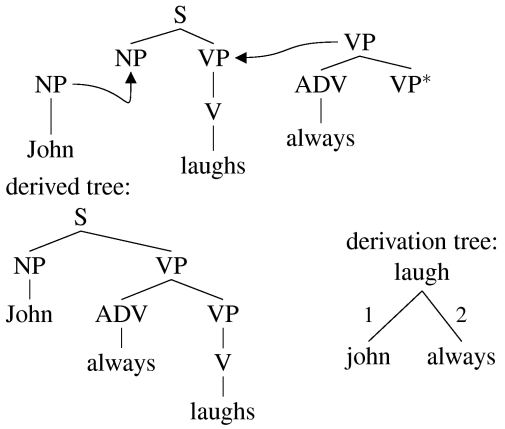


Fig. 1. TAG derivation for *John always laughs*

TAG derivations are represented by derivation trees that record the history of how the elementary trees are put together. A derivation tree is an unordered tree whose nodes are labeled with elements in $I \cup A$ and whose edges are labeled with Gorn addresses of elementary trees[2]. Each edge in a derivation tree stands for an adjunction or a substitution. E.g., the derivation tree in Fig. 1 indicates that the elementary tree for *John* is substituted for the node at address 1 and *always* is adjoined at node address 2 (the fact that the former is an adjunction, the latter a substitution can be inferred from the fact that the node at address 1 is a substitution node while the node at address 2 is an internal node).

In the following, we write a derivation tree D as a directed graph $\langle V, E, r\rangle$ where V is the set of nodes, $E \subset V \times V$ the set of arcs and $r \in V$ the root[3]. For every $v \in V$, $Lab(v)$ gives the node label and for every $\langle v_1, v_2\rangle \in E$, $Lab(\langle v_1, v_2\rangle)$ gives the edge label.

A derived tree is the result of carrying out the substitutions and the adjunctions in a derivation tree, i.e., the derivation tree describes uniquely the derived tree; see again Fig. 1.

The tree language of a TAG G, written $T(G)$, is the set of all trees γ such that γ is derived from an initial tree in G and there is no node v in γ with $f_{OA}(v) = 1$. The string language of G is the set of yields of trees in $L_T(G)$.

2.2. Multicomponent TAG with tree tuples. For a range of linguistic phenomena, multicomponent TAG [4] have been proposed, also called MCTAG for short. The underlying motivation is the desire to split the contribution of a single lexical item (e.g., a verb and its arguments) into several elementary trees. An MCTAG consists of (multi-)sets of elementary trees, called *tree sets*. If an elementary tree from some set is used in a derivation, then all of the remaining trees in the set must be used as well. Several variants of MCTAGs can be found the literature, differing on the specific definition of the derivation process.

The particular MCTAG variant we are concerned with is Tree-Tuple MCTAG with Shared Nodes, TT-MCTAG [1]. TT-MCTAG were introduced to deal with free word order phenomena in languages such as German. An example is (1) where the argument *es* of *reparieren* precedes the argument *der Mann* of *verspricht* and is not adjacent to the predicate it depends on.

(1) ... dass es der Mechaniker zu reparieren verspricht
... that it the mechanic to repair promises
'... that the mechanic promises to repair it'

A TT-MCTAG is slightly different from standard MCTAGs since each elementary tree set contains one specially marked lexicalized tree called the head, and all of the remaining trees in the set function as arguments of the head. Furthermore, in a TT-MCTAG derivation the argument trees must either adjoin directly to their head tree, or they must be linked in the derivation tree to an elementary tree that attaches to the head tree, by means of a chain of adjunctions at root nodes. In other words, in the corresponding TAG derivation tree, the head tree must dominate the argument trees in such a way that all positions on the path between them, except the first one, must be labeled ε. This captures the notion of adjunction under node sharing from [5][4].

Figure 2 shows a TT-MCTAG derivation for (1). Here, the NP_{nom} auxiliary tree adjoins directly to *verspricht* (its head) while the NP_{acc} tree adjoins to the root of a tree that adjoins to the root of a tree that adjoins to *reparieren*.

Definition 2 (TT-MCTAG). A TT-MCTAG is a tuple $G = (V_N, V_T, S, I, A, \mathcal{T})$ where $G_T = (V_N, V_T, S, I, A)$ is an underlying TAG and $\mathcal{T}$ is a finite set of tree tuples of the form $\Gamma = \langle \gamma, \{\beta_1, \ldots, \beta_r\}\rangle$ where $\gamma \in (I \cup A)$ has at least one node with a terminal label, and $\beta_1, \ldots, \beta_n \in A$.

For each $\Gamma = \langle \gamma, \{\beta_1, \ldots, \beta_r\}\rangle \in \mathcal{T}$, we call γ the head tree and the β_j's the argument trees. We informally say that γ and the β_j's belong to Γ, and write $|\Gamma| = r + 1$.

As a remark, an elementary tree γ from the underlying TAG G_T can be found in different tree tuples in G, or there could even be multiple instances of such a tree within the same tree tuple Γ. In these cases, we just treat these tree instances

[2]In this convention, the root address is ε (the empty string) and the jth child of a node with address p has address $p \cdot j$.

[3]Note that the root node of the derived tree can be deduced from the set of arcs.

[4]The intuition is that, if a tree γ' adjoins to some γ, its root in the resulting derived tree somehow belongs both to γ and γ' or, in other words, is shared by them. A further tree β adjoining to this node can then be considered as adjoining to γ, not only to γ' as in standard TAG. Note that we assume that foot nodes do not allow adjunctions, otherwise node sharing would also apply to them.

as distinct trees that are isomorphic and have identical labels. For a given argument tree β in Γ, $h(\beta)$ denotes the head of β in Γ. For a given $\gamma \in I \cup A$, $a(\gamma)$ denotes the set of argument trees of γ, if there are any, or the empty set otherwise. Furthermore, for a given TT-MCTAG G, $H(G)$ is the set of head trees and $A(G)$ is the set of argument trees. Finally, a node v in a derivation tree for G with $Lab(v) = \gamma$ is called a γ-node.

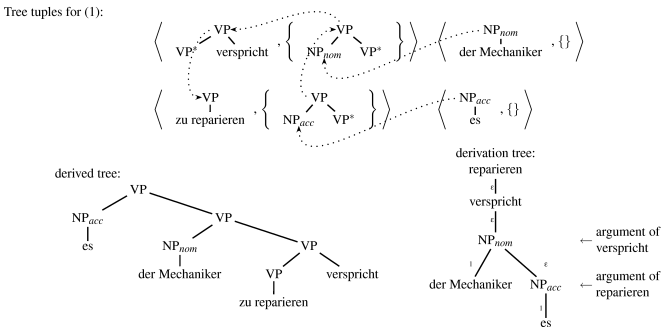


Fig. 2. TT-MCTAG analysis of (1)

Definition 3 (TT-MCTAG derivation).
Let $G = (V_N, V_T, S, I, A, \mathcal{T})$ be some TT-MCTAG. A derivation tree $D = \langle V, E, r \rangle$ in the underlying TAG G_T is licensed in G if and only if the following conditions (MC) and (SN-TTL) are both satisfied.

- **(MC)**: For all Γ from G and for all γ_1, γ_2 in Γ, we have $|\{v \,|\, v \in V,\, Lab(v) = \gamma_1\}| = |\{v \,|\, v \in V,\, Lab(v) = \gamma_2\}|$.
- **(SN-TTL)**: For all $\beta \in A(G)$ and $n \geq 1$, let $v_1, \ldots, v_n \in V$ be pairwise different $h(\beta)$-nodes, $1 \leq i \leq n$. Then there are pairwise different β-nodes $u_1, \ldots, u_n \in V$, $1 \leq i \leq n$. Furthermore, for $1 \leq i \leq n$, either $\langle v_i, u_i \rangle \in E$, or else there are $u_{i,1}, \ldots, u_{i,k}$, $k \geq 2$, with auxiliary tree labels, such that $u_i = u_{i,k}$, $\langle v_i, u_{i,1} \rangle \in E$ and, for $1 \leq j \leq k-1$, $\langle u_{i,j}, u_{i,j+1} \rangle \in E$ with $Lab(\langle u_{i,j}, u_{i,j+1} \rangle) = \varepsilon$.

The separation between (MC) and (SN-TTL) in definition 3 is motivated by the desire to separate the multicomponent property that TT-MCTAG shares with a range of related formalisms (e.g., tree-local and set-local MCTAG, VectorTAG, etc.) from the notion of tree-locality with shared nodes that is peculiar to TT-MCTAG.

As already mentioned, TT-MCTAG has been proposed to deal with free word order languages. An example from German was given in Fig. 2. For a more extended account of German word order using TT-MCTAG see [1] and [6].

TT-MCTAG can be further restricted, such that at each point of the derivation the number of pending β-trees is at most k. This subclass is called k-TT-MCTAG.

Definition 4 (k-TT-MCTAG). A TT-MCTAG $G = \langle I, A, N, T, \mathcal{A} \rangle$ is of rank k (or a k-TT-MCTAG for short) iff for each derivation tree D licenced in G:

(TT-k) There are no nodes n, $h_0, \ldots, h_k$, $a_0, \ldots, a_k$ in D such that the label of a_i is an argument tree of the label of h_i and $\langle h_i, n \rangle, \langle n, a_i \rangle \in \mathcal{P}_D^+$ for $0 \leq i \leq k$.

Both, TT-MCTAG and the restricted k-TT-MCTAG, generate only polynomial languages [7, 8].

3. Transforming TT-MCTAG into RCG

The central idea of our parsing strategy is to use RCG [9, 10] as a pivot formalism, for it has several interesting properties: first, RCGs are known to describe exactly the class PTIME of languages parsable in polynomial time [11], secondly, their generative capacity lies beyond *Linear Context-Free Rewriting Systems* (LCRFS, [4]). Therehere exists a restricted class of RCGs called *simple RCGs*, which has been shown to be equivalent to LCFRS [12]. Actually, simple RCG and LCFRS are more or less syntactic variants of each other.

Informally speaking, RCGs are grammars that rewrite predicates ranging over parts of the input string by other predicates. E.g., a clause $S(aXb) \rightarrow S(X)$ signifies that S is true for a part of the input if this part starts with an a, ends with a b, and if, furthermore, S is also true for the part between a and b.

In the following, a formal definition of RCG is given, and then the algorithm used to transform a TAG (respectively a TT-MCTAG) into an equivalent RCG is presented.

3.1. Range Concatenation Grammar. In this paper, by range concatenation grammar, we always mean Positive Range Concatenation Grammar, since this is the variant commonly considered in the above mentioned applications. Negative RCG allows for negative predicate calls of the form $\overline{A(\alpha_1, \ldots, \alpha_n)}$. Such a predicate is meant to recognize the complement language of its positive counterpart. See [10] for details.

Definition 5 (Range Concatenation Grammar). A range concatenation grammar (RCG) is a 5-tuple $G = \langle N, T, V, P, S \rangle$ where

- N is a finite set of predicate names with an arity function dim: $N \rightarrow \mathbb{N} \setminus \{0\}$,
- T and V are finite sets of terminals and variables.
- P is a finite set of clauses of the form
$$\psi_0 \rightarrow \psi_1 \ldots \psi_m$$
where $m \geq 0$ (if $m = 0$, then the right-hand-side of the clause is empty) and each of the $\psi_i, 0 \leq i \leq m$, is a predicate of the form $A_i(\alpha_1, \ldots, \alpha_{dim(A_i)})$ with $A_i \in N$ and $\alpha_j \in (T \cup V)^*$ for $1 \leq j \leq dim(A_i)$.
As a shorthand notation for $A_i(\alpha_1, \ldots, \alpha_{dim(A_i)})$, we use the vector-based notation $A_i(\vec{\alpha})$.
- $S \in N$ is the start predicate name with $dim(S) = 1$.

We assume our RCGs to be ε-free, i.e., not to contain lefthand side arguments ε. This can be done without loss of generality [12].

To illustrate this definition, see the example RCG given in Fig. 3.

$$G = \langle \{S, eq\}, \{a\}, \{X, Y\}, P, S \rangle \text{ with } P =$$
$$\begin{array}{lll} \{ \; S(XY) & \rightarrow & S(X)\ eq(X,Y), \\ S(a) & \rightarrow & \varepsilon, \\ eq(aX, aY) & \rightarrow & eq(X,Y), \\ eq(a,a) & \rightarrow & \varepsilon \quad \} \end{array}$$

Fig. 3. Example of an RCG

Central to RCGs is the notion of ranges on strings.

Definition 6 (Ranges). For every $w \in T^*$, where $w = w_1 \dots w_n$ with $w_i \in T$ for $1 \leq i \leq n$, we define:

- $Pos(w) := \{0, \dots, n\}$.
- A pair $\langle l, r \rangle \in Pos(w) \times Pos(w)$ with $l \leq r$ is a range in w. Its yield $\langle l, r \rangle(w)$ is the substring $w_{l+1} \dots w_r$.
- For two ranges $\rho_1 = \langle l_1, r_1 \rangle, \rho_2 = \langle l_2, r_2 \rangle$: if $r_1 = l_2$, then $\rho_1 \cdot \rho_2 = \langle l_1, r_2 \rangle$; otherwise $\rho_1 \cdot \rho_2$ is undefined.

Definition 7 (Range vectors). For a given $w \in T^*$, we call a vector $\phi = (\langle x_1, y_1 \rangle, \dots, \langle x_k, y_k \rangle)$ a range vector of dimension k in w if $\langle x_i, y_i \rangle$ is a range in w for $1 \leq i \leq k$. $\phi(i).l$ (resp. $\phi(i).r$) denotes then the first (resp. second) component of the ith element of ϕ, that is x_i (resp. y_i).

In order to instantiate a clause of the grammar, we need to find ranges for all variables in the clause and for all occurrences of terminals. For convenience, we assume the variables in a clause, the occurrences of terminals and the occurrences of empty arguments ε to be equipped with distinct subscript indices, starting with 1 and ordered from left to right (where for variables, only the first occurrence is relevant for this order). We then introduce a function $\Upsilon : P \to \mathbb{N}$ that gives the maximal index in a clause. Furthermore, we define $\Upsilon(c, x)$ for a given clause c and x a variable or an occurrence of a terminal or an empty argument ε as the index of x in c.

Definition 8 (Clause instantiation). An instantiation of some clause $c \in P$ with $\Upsilon(c) = j$ wrt. to some string w is given by a range vector ϕ with $dim(\phi) = j$. Applying ϕ to a predicate $A(\vec{\alpha})$ in c maps all occurrences of $x \in (T \cup V \cup \{\varepsilon\})$ with $\Upsilon(c, x) = i$ in $\vec{\alpha}$ to $\phi(i)$. If the result is defined (i.e., the images of adjacent variables can be concatenated), it is called an instantiated predicate and the result of applying ϕ to all predicates in c, if defined, is called an instantiated clause.

An RCG derivation consists of rewriting instantiated predicates applying instantiated clauses:

Definition 9 (Derivation). Given an RCG G and an input string w, we define a relation $\Rightarrow_{G,w}$ called derive on strings of instantiated predicates the following way. Let Γ_1, Γ_2 be strings of instantiated predicates. If $A_0(\vec{\alpha_0}) \to A_1(\vec{\alpha_1}) \dots A_m(\vec{\alpha_m})$ is the instantiation of some clause $c \in P_G$, then $\Gamma_1 A_0(\vec{\alpha_0}) \Gamma_2 \Rightarrow_{G,w} \Gamma_1 A_1(\vec{\alpha_1}) \dots A_m(\vec{\alpha_m}) \Gamma_2$.

Intuitively, if the left-hand-side (LHS) of an instantiated clause occurs in some string of instantiated predicates, it may be replaced by its righthand side.

Definition 10 (Language). The language of an RCG G is the set of strings that can be reduced to the empty word: $L(G) = \{w \mid S(\langle 0, |w| \rangle) \stackrel{+}{\Rightarrow}_{G,w} \varepsilon\}$.

The expressive power of RCG lies beyond mild context-sensitivity. Let us consider the RCG G defined in Fig. 3. It is easy to see that $L(G) = \{a^{2^n} \mid n \geq 0\}$. This language is obviously not mildly context-sensitive as it does not have the constant growth property.

As an example of RCG derivation, let us consider the RCG G' defined as follows. $G' = \langle \{S, A, B\}, \{a, b\}, \{X, Y, Z\}, S, P \rangle$ where $P = \{S(XYZ) \to A(X, Z)B(Y), A(aX, aY) \to A(X, Y), B(bX) \to B(X), A(\varepsilon, \varepsilon) \to \varepsilon, \ B(\varepsilon) \to \varepsilon\}$.

The string language generated by G' is $L(G') = \{a^n b^k a^n \mid k, n \in \mathbb{N}\}$.

Consider the input string $w = aabaa$. To parse this string, first, the clause whose LHS predicate is S is instantiated as follows. $S(\langle 0,5 \rangle) \Rightarrow A(\langle 0,2 \rangle, \langle 3,5 \rangle)B(\langle 2,3 \rangle)$.

$$\begin{array}{ccccccccc} S(X & & Y & & Z) & \to & A(X, & Z) & B(Y) \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & \downarrow & \downarrow \\ \langle 0,2 \rangle & & \langle 2,3 \rangle & & \langle 3,5 \rangle & & \langle 0,2 \rangle & \langle 3,5 \rangle & \langle 2,3 \rangle \\ aa & & b & & aa & & aa & aa & b \end{array}$$

Secondly, we replace the succesfully instantiated LHS with its RHS, this gives us the following instantiations: $A(\langle 0,2 \rangle, \langle 3,5 \rangle) \Rightarrow A(\langle 1,2 \rangle, \langle 4,5 \rangle) \Rightarrow A(\langle 2,2 \rangle, \langle 5,5 \rangle) \Rightarrow \varepsilon$.

$$\begin{array}{ccccccccc} A(a & & X & & a & & Y) & \to & A(X, & Y) \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \downarrow \\ \langle 0,1 \rangle & & \langle 1,2 \rangle & & \langle 3,4 \rangle & & \langle 4,5 \rangle & & \langle 1,2 \rangle & \langle 4,5 \rangle \\ a & & a & & a & & a & & a & a \end{array}$$

$$\begin{array}{ccccccccc} A(a & & X & & a & & Y) & \to & A(X, & Y) \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \downarrow \\ \langle 1,2 \rangle & & \langle 2,2 \rangle & & \langle 4,5 \rangle & & \langle 5,5 \rangle & & \langle 2,2 \rangle & \langle 5,5 \rangle \\ a & & \varepsilon & & a & & \varepsilon & & \varepsilon & \varepsilon \end{array}$$

and $A(\varepsilon, \varepsilon) \to \varepsilon$

We still have to instantiate the second predicate of the starting clause: $B(\langle 2,3 \rangle) \Rightarrow B(\langle 3,3 \rangle) \Rightarrow \varepsilon$.

$$\begin{array}{ccccc} B(b & & X) & \to & B(X) \\ \downarrow & & \downarrow & & \downarrow \\ \langle 2,3 \rangle & & \langle 3,3 \rangle & & \langle 3,3 \rangle \\ b & & \varepsilon & & \varepsilon \end{array} \qquad \text{and } B(\varepsilon) \to \varepsilon$$

Since ε can be derived, $w \in L(G')$.

Definition 11 (Simple RCG). An RCG is called simple RCG if it is non-combinatorial, linear and non-erasing.

- An RCG is said to be non-combinatorial when all arguments of its RHS predicate are made of single variables (i.e., concatenations of variables are not allowed within an argument of a RHS predicate),
- An RCG is said to be linear when no variable appears more than once within either the LHS or the RHS of a clause,
- An RCG is said to be non-erasing when all variables appearing in a RHS of a clause also appear in its LHS and vice versa.

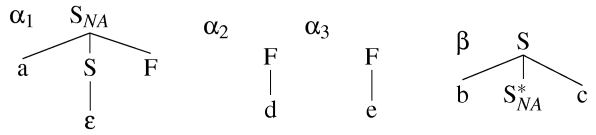
To sum up, a simple RCG is a set of clauses where all arguments of the RHS predicate are made of single variables only and where every variable occurs exactly once both in the LHS and in the RHS. Note that the RCG G generating $\{a^n b^k a^n \mid k, n \in \mathbb{N}\}$ introduced above is, for example, a simple RCG.

To encode TAGs and TT-MCTAGs, as we shall see in the next sections, only simple RCGs are needed.

3.2. Converting a TAG into an equivalent RCG. The idea of the TAG-to-RCG transformation has been proposed by Boullier [9, 13] and is the following: the RCG contains clauses with LHS predicates of the form $\langle\alpha\rangle(X)$ and $\langle\beta\rangle(L,R)$ for initial and auxiliary trees respectively.

More precisely, an RCG clause is created for every single elementary tree of the TAG. If the tree is an initial tree α, the LHS predicate of the clause is named $\langle\alpha\rangle$, and has a single argument X referring to the substring of the input string covered by α (including all trees that may have been combined with α via tree rewriting). If the tree is an auxiliary tree β, the LHS predicate of the clause named $\langle\beta\rangle$, has two arguments L and R covering those parts of the input string represented by the yields of β (including all trees added to β) that are respectively to the left and the right of the foot node of β.

TAG:



Equivalent RCG:

$S(X) \rightarrow \langle\alpha_1\rangle(X) \quad S(X) \rightarrow \langle\alpha_2\rangle(X) \quad S(X) \rightarrow \langle\alpha_3\rangle(X)$
(every word in the language is the yield of an $\alpha \in I$, for S is the axiom of our TAG)

$\langle\alpha_1\rangle(aX) \rightarrow sub_F(X)$
(the yield of α_1 can be a followed by the tree that substitutes at F, if there is no adjunction at the inner S node)

$sub_F(X) \rightarrow \langle\alpha_2\rangle(X) \quad sub_F(X) \rightarrow \langle\alpha_3\rangle(X)$
(the tree that substitutes at F is either α_2 or α_3)

$\langle\alpha_1\rangle(aB_1B_2X) \rightarrow adj_S(B_1,B_2)\ sub_F(X)$
(the yield of α_1 can be a followed by the left and right parts of the adjunction at S followed by the yield of the tree substituted at F)

$adj_S(L,R) \rightarrow \langle\beta\rangle(L,R)$
(β can adjoin at a node labelled S; then the yield is the left and the right parts of β)

$\langle\beta\rangle(B_1b,cB_2) \rightarrow adj_S(B_1,B_2)$
(β can host an adjunction on its root; then the left part is the left part of the adjoined tree followed by b; the right part is c followed by the right part of the adjoined tree)

$\langle\alpha_2\rangle(d) \rightarrow \varepsilon \quad \langle\alpha_3\rangle(e) \rightarrow \varepsilon \quad \langle\beta\rangle(b,c) \rightarrow \varepsilon$
(the yields of α_2, α_3 and β can be d, e and the pair b (left) and c (right) resp.)

Fig. 4. A TAG and an equivalent RCG

The clauses in the RCG reduce the argument(s) of these predicates by identifying those parts that come from the elementary tree α/β itself and those parts that come from one of the elementary trees added by substitution or adjunction. In other words, the RHS predicates of the above clauses are *branching* predicates relating arguments of the LSH predicate (i.e., parts of the input string) with trees they are included in the yield of. As an example, consider Fig. 4[5]. The clause $\langle\alpha_1\rangle(aX) \rightarrow sub_F(X)$ encodes the fact that the yield of the tree α_1 contains the terminal symbol a followed by symbols included in the yield of a tree that has been substituted at node F in α_1. Then, $sub_F(X) \rightarrow \langle\alpha_2\rangle(X)$ states that the tree α_2 can substitute at a node labelled F.

3.3. Converting a TT-MCTAG into an equivalent RCG. For the transformation from TT-MCTAG into RCG, we use the same idea as for TAG. There are predicates $\langle\gamma...\rangle$ for the elementary trees (not the tuples) that characterize the contribution of γ. We enrich these predicates in a way that allows to keep track of the "still to adjoin" argument trees and constrain thereby further the RCG clauses. The pending arguments are encoded in an unordered list (called List of Pending Arguments – LPA) that is part of the predicate name. The yield of a predicate corresponding to a tree γ contains not only γ and its arguments but also arguments of predicates that are higher in the derivation tree and that are adjoined below γ via node sharing. In addition, we use branching predicates *adj* and *sub* that allow computation of the possible adjunctions or substitutions at a given node in a separate clause, thus reducing the number of clauses.

To sum up, we handle three types of clauses, depending on the form of their left-hand side predicate:

1. $\langle\gamma, LPA\rangle$ where γ is a tree (not a tuple) from the TT-MCTAG, and *LPA* is the list of argument trees waiting for adjunction (used to constraint the derivation, *i.e.*, the form of the RHS predicates, as will be shown below),
2. $\langle adj, \gamma, dot, LPA\rangle$ where *dot* is the node of the tree γ which may hosts an adjunction. The LPA contains the argument trees of γ (plus the argument trees currently waiting for adjunction, if *dot* is the root node), provided these trees have the necessary node labels to adjoin in *dot*,
3. $\langle sub, \gamma, dot\rangle$ where *dot* is a substitution node of γ.

In order to give the transformation algorithm, we need to define the decoration string of a tree τ:

Definition 12 (Decoration string). Given a tree τ having n adjunction nodes $an_i, n \geq i \geq 0$, and m substitution nodes $sn_j, m \geq j \geq 0$, we associate each node an_i with a pair of variables L_{an_i}, R_{an_i}, and each node sn_j with a variable X_{sn_j}.

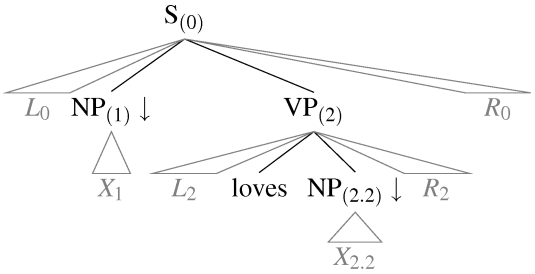
L_{an_i}(resp. R_{an_i}) represents the range of the input string corresponding to the left (resp. right) yield of the tree adjoined in an_i (if any). By left (resp. right) yield, we mean the yield that is situated on the left (resp. right) of the foot node of the adjoined tree.

X_{sn_j} refers to the range of the input string corresponding to the yield of the tree substituted in sn_j.

Figure 5 shows an example of the decoration string of an elementary tree[6].

[5] In this figure, the node label S_{NA} corresponds to an extension of the label S with the additional *Null-Adjunction* constraint. In other words, a node labelled S_{NA} is a node with label S, and which cannot host any adjunction.

[6] The indices associated with the nodes are their Gorn addresses.



Decoration string : $L_0\,X_1\,L_2$ *loves* $X_{2.2}\,R_2\,R_0$

Fig. 5. A TAG tree and its decoration string

The conversion algorithm. From the above informations, we can now detail the TT-MCTAG-to-RCG conversion algorithm. Note that this algorithm is based on an agenda, which creates new clauses from existing ones, until a given point has been reached. Usually, this limit is defined in terms of length of the list of pending arguments. Therefore, strictly speaking, this algorithm converts *k*-TT-MCTAG (introduced in Sec. 2) into equivalent RCG.

1. We build clauses whose LHS are the start predicate (of arity 1), and whose RHS are made of single predicates of the form $\langle \alpha, \emptyset \rangle$, where α is a head tree of a tuple of the input TT-MCTAG, such that the root of α is labelled with the axiom of the TT-MCTAG.
2. For each tree γ (initial or auxiliary tree) in the input TT-MCTAG, let us call σ_γ its decoration string, and L_p, R_p the variables associated with the node at position p, if it is an adjunction node, or X_p if it is a substitution node. Let $p_1, ..., p_k$ (resp. $p_{k+1}, ..., p_n$) be the adjunction nodes (resp. substitution nodes) of γ. We then build the following clauses: $\langle \gamma, LPA \rangle(\sigma_\gamma) \rightarrow \langle adj, \gamma, p_1, LPA_{p_1} \rangle(L_{p_1}, R_{p_1}) \ldots \langle adj, \gamma, p_k, LPA_{p_k} \rangle (L_{p_k}, R_{p_k}) \langle sub, \gamma, p_{k+1} \rangle(X_{p_{k+1}}) \ldots \langle sub, \gamma, p_l \rangle(X_{p_l})$ such that
   - If $LPA \neq \emptyset$, then $\varepsilon \in \{p_1, \ldots, p_k\}$ (*i.e.*, one of the adjunction nodes is the root) and $LPA \subseteq LPA_\varepsilon$, and
   - $\bigcup_{i=0}^{k} LPA_{p_i} = LPA \cup \Gamma(\gamma)$ where $\Gamma(\gamma)$ is either the set of arguments of γ (if γ is a head tree) or (if γ is an argument itself), the empty set.
3. For all predicates $\langle adj, \gamma, dot, LPA \rangle$ computed at step 2., the RCG contains all clauses $\langle adj, \gamma, dot, LPA \rangle(L, R) \rightarrow \langle \gamma', LPA' \rangle(L, R)$ such that γ' can be adjoined at position *dot* in γ and
   - either $\gamma' \in LPA$ and $LPA' = LPA \setminus \{\gamma'\}$,
   - or $\gamma' \notin LPA$, γ' is a head tree, and $LPA' = LPA$.
4. For all predicates $\langle adj, \gamma, dot, \emptyset \rangle$ where *dot* in γ is not a node of type *mandatory adjunction*, the RCG contains a clause $\langle adj, \gamma, dot, \emptyset \rangle(\varepsilon, \varepsilon) \rightarrow \varepsilon$.
5. For all predicates $\langle sub, \gamma, dot \rangle$ and all γ' that can be substituted into position *dot* in γ the RCG contains a clause $\langle sub, \gamma, dot \rangle(X) \rightarrow \langle \gamma', \emptyset \rangle(X)$.

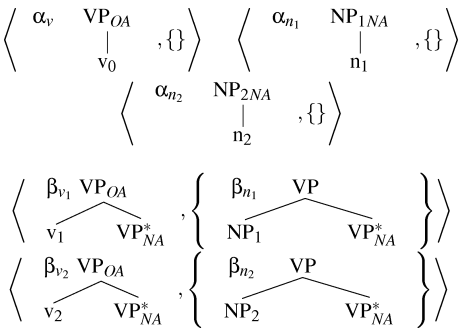


Fig. 6. Toy TT-MCTAG

As a first illustration of this algorithm, let us take the TT-MCTAG displayed in Fig. 6. From this TT-MCTAG, the following RCG clauses are built (amongst others):

$\langle \alpha_v, \emptyset \rangle(L\ v_0\ R) \rightarrow \langle adj, \alpha_v, \varepsilon, \emptyset \rangle(L, R)$
(a single adjunction at the root node ε)

$\langle adj, \alpha_v, \varepsilon, \emptyset \rangle(L, R) \rightarrow \langle \beta_{v_1}, \emptyset \rangle(L, R)$
$\langle adj, \alpha_v, \varepsilon, \emptyset \rangle(L, R) \rightarrow \langle \beta_{v_2}, \emptyset \rangle(L, R)$
(β_{v_1} or β_{v_2} can adjoin at ε in α_v, LPA (here empty) is passed to the RHS)

$\langle \beta_{v_1}, \emptyset \rangle(L\ v_1, R) \rightarrow \langle adj, \beta_{v_1}, \varepsilon, \{\beta_{n_1}\} \rangle(L, R)$
(in β_{v_1}, there is a unique adjunction node whose address is ε; the argument tree is fed to the new LPA)

$\langle adj, \beta_{v_1}, \varepsilon, \{\beta_{n_1}\} \rangle(L, R) \rightarrow \langle \beta_{n_1}, \emptyset \rangle(L, R)$
$\langle adj, \beta_{v_1}, \varepsilon, \{\beta_{n_1}\} \rangle(L, R) \rightarrow \langle \beta_{v_1}, \{\beta_{n_1}\} \rangle(L, R)$
$\langle adj, \beta_{v_1}, \varepsilon, \{\beta_{n_1}\} \rangle(L, R) \rightarrow \langle \beta_{v_2}, \{\beta_{n_1}\} \rangle(L, R)$
(either β_{n_1} is adjoined and removed from the LPA, or another tree (β_{v_1} or β_{v_2}) is adjoined)

$\langle \beta_{v_1}, \{\beta_{n_1}\} \rangle(L\ v_1, R) \rightarrow \langle adj, \beta_{v_1}, \varepsilon, \{\beta_{n_1}, \beta_{n_1}\} \rangle(L, R)$
(here again, there is a unique adjunction node in β_{v_1}; the argument tree β_{n_1} is added to the LPA)

$\langle \beta_{n_1}, \emptyset \rangle(L\ X, R) \rightarrow \langle adj, \beta_{n_1}, \varepsilon, \emptyset \rangle(L, R)\ \ \langle sub, \beta_{n_1}, 1, \rangle(X)$
(root adjunction and substitution at node 1 in β_{n_1})

$\langle adj, \beta_{n_1}, \varepsilon, \emptyset \rangle(\varepsilon, \varepsilon) \rightarrow \varepsilon$
(the root adjunction in β_{n_1} is not mandatory provided the LPA is empty)

$\langle sub, \beta_{n_1}, 1, \rangle(X) \rightarrow \langle \alpha_{n_1}, \emptyset \rangle(X)$
(substitution inserting α_{n_1} at node of address 1 in β_{n_1})

$\langle \alpha_{n_1}, \emptyset \rangle(n_1) \rightarrow \varepsilon$
(no adjunction or substitution in α_{n_1})

As a second example, see Fig. 7, which gives some of the clauses built via our algorithm from the TT-MCTAG in Fig. 2. The first clause here states that the yield of the initial tree α_{rep} consists of the left and right parts of the root-adjoining tree wrapped around *zu reparieren*. The *adj* predicate takes care of the adjunction at the root of α_{rep} (node whose Gorn address is ε). This *adj* predicate also encodes the fact that the list of pending arguments contains β_{acc}, the argument tree of α_{rep}. According to the second and third clauses, we can adjoin either β_{acc} (while removing it from the list of pending arguments) or some new auxiliary tree β_v. Then, we have

a clause defining how β_{acc} itself can be used in a derivation, i.e. which part of its yield comes from some adjunction and which from some substitution. We thus compute new clauses, taking into account the evolution of the list of pending arguments. This computation is known to stop as we consider consider k-TT-MCTAG, for which we can restrict the length of the list of pending arguments, and thus stop the production of new clauses.

$$\langle \alpha_{rep}, \emptyset \rangle (L\ zu\ reparieren\ R) \rightarrow \langle adj, \alpha_{rep}, \varepsilon, \{\beta_{acc}\} \rangle (L,R)$$
$$\langle adj, \alpha_{rep}, \varepsilon, \{\beta_{acc}\} \rangle (L,R) \rightarrow \langle \beta_{acc}, \emptyset \rangle (L,R)$$
$$\langle adj, \alpha_{rep}, \varepsilon, \{\beta_{acc}\} \rangle (L,R) \rightarrow \langle \beta_{v}, \{\beta_{acc}\} \rangle (L,R)$$
$$\langle \beta_{acc}, \emptyset \rangle (L\ X\ R) \rightarrow \langle adj, \beta_{acc}, \varepsilon, \emptyset \rangle (L,R) \langle sub, \beta_{acc}, 1 \rangle (X)$$
$$\langle sub, \beta_{acc}, 1 \rangle (X) \rightarrow \langle \alpha_{es}, \emptyset \rangle (X) \qquad \langle \alpha_{es}, \emptyset \rangle (es) \rightarrow \varepsilon$$
$$\langle \beta_{v}, \{\beta_{acc}\} \rangle (L, verspricht\ R) \rightarrow \langle adj, \beta_{v}, \varepsilon, \{\beta_{nom}, \beta_{acc}\} \rangle (L,R)$$
$$\ldots$$

Fig. 7. Some clauses of the RCG correponding to the TT-MCTAG in Fig. 2

Note that, from this transformation algorithm, we obtain the following result: for each k-TT-MCTAG G, there exists an equivalent simple RCG G' such that $L(G) = L(G')$, thus k-TT-MCTAG is mildly context-sensitive (recall that simple RCG are equivalent to LCFRS, which are mildly context-sensitive).

The conversion can be optimized in order to decrease the number of RCG predicates by using branching predicates of the form $\langle adj, cat, LPA \rangle$ and $\langle sub, cat \rangle$ that describe possible adjunctions/substitutions at nodes with category cat.

4. RCG parsing

The input sentence is parsed using the RCG computed from the input TT-MCTAG via the conversion algorithm introduced in the previous section. Note that the TT-MCTAG to RCG transformation is applied to a subgrammar selected from the input sentence since the cost of the conversion depends heavily on the size of the grammar (all licensed adjunctions have to be computed while taking into account the state of the list of pending arguments)[7].

The RCGs one obtains from transforming a k-TT-MCTAG are always ordered simple RCGs. A simple RCG is *ordered* if the order of variables is the same in the lhs and rhs predicates of a clause. In particular, this means that for every clause instantiation, the order of arguments of a predicate is the same as the order of the correponding ranges in the input string.

In the spirit of delivering a multi-formalism archtitecture, our system contains implementations of different algorithms, namely of the Earley-style parsing algorithm for simple RCG, presented in [14], and of two algorithms for the full class of RCG [10, 15].

4.1. Preliminaries. In order to formulate our parsing algorithms, besides range vectors, we also introduce range constraint vectors. These are vectors of pairs of range boundary variables together with a set of constraints on these variables.

Definition 13 (Range constraint vectors). Let $V_r = \{r_1, r_2, \ldots\}$ be a set of range boundary variables.

A range constraint vector of dimension k is a pair $\langle \vec{\rho}, C \rangle$ where

- $\rho \in (V_r^2)^k$; we define $V_r(\rho)$ as the set of range boundary variables occurring in $\vec{\rho}$.
- C is a set of constraints c_r that have one of the following forms:
  $r_1 = r_2, \quad k = r_1, \quad r_1 + k = r_2, \quad k \leq r_1, \quad r_1 \leq k,$
  $r_1 \leq r_2$ or $r_1 + k \leq r_2$ for $r_1, r_2 \in V_r(\rho)$ and $k \in \mathbb{N}$.

We say that a range vector ϕ satisfies a range constraint vector $\langle \rho, C \rangle$ iff ϕ and ρ are of the same dimension k and there is a function $f : V_r \rightarrow \mathbb{N}$ that maps $\rho(i).l$ to $\phi(i).l$ and $\rho(i).r$ to $\phi(i).r$ for all $1 \leq i \leq k$ such that all constraints in C are satisfied. Furthermore, we say that a range constraint vector $\langle \rho, C \rangle$ is satisfiable iff there exists a range vector ϕ that satisfies it.

Definition 14 (Range constraint vector of a clause). For every clause c, we define its range constraint vector $\langle \rho, C \rangle$ wrt. to a w with $|w| = n$ as follows:

- ρ has dimension $\Upsilon(c)$ and all range boundary variables in ρ are pairwise different.
- For all $\langle r_1, r_2 \rangle \in \rho$:
  $0 \leq r_1, \quad r_1 \leq r_2, \quad r_2 \leq n \in C.$
  For all occurrences x of terminals in c with $i = \Upsilon(c,x)$:
  $\rho(i).l + 1 = \rho(i).r \in C.$
  For all x, y that are variables or occurrences of terminals in c such that xy is a substring of one of the arguments in c:
  $\rho(\Upsilon(c,x)).r = \rho(\Upsilon(c,y)).l \in C.$
  These are all constraints in C.

Intuitively, a range constraint vector of some clause captures all information about boundaries forming a range, ranges containing only a single terminal, and about adjacent variables/terminal occurrences in the clause.

4.2. Parsing simple RCG. The general idea is that we process the arguments of the lefthand sides of clauses incrementally, starting from an S-clause. Whenever we reach a variable, we move into the clause of the corresponding rhs predicate (**predict** or **resume**). Whenever we reach the end of an argument, we **suspend** this clause and move into the parent clause that has called the current one.

Our items have the form

$$[A(\vec{\phi}) \rightarrow A_1(\vec{\phi_1}) \ldots A_m(\vec{\phi_m}), pos, \langle i, j \rangle, \vec{\rho}],$$

where

- $A(\vec{\phi}) \rightarrow A_1(\vec{\phi_1}) \ldots A_m(\vec{\phi_m})$ is a clause;
- $pos \in \{0, \ldots, n\}$ is the position up to which we have processed theinput;
- $\langle i, j \rangle \in \mathbb{N}^2$ marks the position of our dot in the arguments of the predicate A: $\langle i, j \rangle$ indicates that we have processed the arguments up to the jth element of the ith argument;

[7] We do not have a proof of complexity of the conversion algorithm yet, but we conject that it is exponential in the size of the grammar since the adjunctions to be predicted depend on the adjunctions predicted so far and on the auxiliary trees adjoinable at a given node.

- $\vec{\rho}$ is an range vector containing the bindings of the variables occurring in the clause. $\vec{\rho}(i)$ is the range the i-th variable in the lefthand side is bound to.) When first predicting a clause, $\vec{\rho}$ is initialized with a vector containing only symbols "?" for "unknown". We call such a vector (of appropriate arity) $\vec{\rho}_{init}$. We write $\vec{\rho}(X)$ for the range bound to the variable X in $\vec{\rho}$.

Applying a range vector $\vec{\rho}$ containing variable bindings to a $\alpha \in (T \cup V)^*$ means mapping every variable X to $\vec{\rho}(X)$ and concatenating adjacent ranges.

We say that two $\alpha_1, \alpha_2 \in (T \cup R)^*$ where R is the set of ranges over w are compatible iff we can find instantiations $f_1, f_2 : (T \cup R) \to R$ such that

- $f_i(r) = r$ for every $r \in R$ and for $1 \leq i \leq 2$,
- $f_i(t) = r$ with $r(w) = t$ for $1 \leq i \leq 2$,
- $f_i(xy) = f_i(x) f_i(y)$ for all $x, y \in T \cup R$ for $1 \leq i \leq 2$ and
- $f_1(\alpha_1) = f_2(\alpha_2)$.

Note that in all the compatibility checks needed below, one of the α_1, α_2 is a range, i.e., does not contain terminals.

We start by predicting the S-predicate:

$$\frac{}{[S(\vec{\phi}) \to \vec{\Phi}, 0, \langle 1,0\rangle, \vec{\rho}_{init}]} S(\vec{\phi}) \to \vec{\Phi} \text{ a clause.}$$

Scan: Whenever the next symbol after the dot is the next terminal in the input, we can scan it:

$$\frac{[A(\vec{\phi}) \to \vec{\Phi}, pos, \langle i,j\rangle, \vec{\rho}]}{[A(\vec{\phi}) \to \vec{\Phi}, pos+1, \langle i, j+1\rangle, \vec{\rho}]} \vec{\phi}(i, j+1) = w_{pos+1}.$$

Scan-ε: Empty arguments have to be treated in the following way.

$$\frac{[A(\vec{\phi}) \to \vec{\Phi}, pos, \langle i,j\rangle, \vec{\rho}]}{[A(\vec{\phi}) \to \vec{\Phi}, pos, \langle i, j+1\rangle, \vec{\rho}]} \vec{\phi}(i, j) = \varepsilon.$$

Predict: Whenever our dot is left of a variable that is the first argument of some righthand side predicate B, we predict new B-clauses:

$$\frac{[A(\vec{\phi}) \to \ldots B(X, \ldots) \ldots, pos, \langle i,j\rangle, \vec{\rho}_A]}{[B(\vec{\psi}) \to \vec{\Psi}, pos, \langle 1,0\rangle, \vec{\rho}_{init}]},$$

where $\vec{\phi}(i, j+1) = X$ and $B(\vec{\psi}) \to \vec{\Psi}$ is a clause.

Suspend: Whenever we arrive at the end of an argument, we suspend the processing of this clause and we go back to the item that was used to predict it.

$$\frac{[B(\vec{\psi}) \to \vec{\Psi}, pos', \langle i,j\rangle, \vec{\rho}_B], \quad [A(\vec{\phi}) \to \ldots B(\vec{\xi}) \ldots, pos, \langle k,l\rangle, \vec{\rho}_A]}{[A(\vec{\phi}) \to \ldots B(\vec{\xi}) \ldots, pos', \langle k, l+1\rangle, \vec{\rho}]},$$

where

- $|\vec{\psi}(i)| = j$ (the ith argument has length j and has therefore been completely processed),
- $\vec{\rho}_B(\vec{\psi}(i))$ (contains ranges and terminals) compatible with the range $\langle pos, pos'\rangle$,
- and for all $1 \leq h < i$: $\vec{\rho}_B(\vec{\psi}(h))$ (contains ranges and terminals) compatible with $\vec{\rho}_A(\vec{\xi}(h))$ (a range).

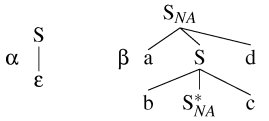
$\vec{\rho}$ is $\vec{\rho}_A$ updated with $\vec{\rho}_A(\vec{\xi}(i)) = \langle pos, pos'\rangle$.

Resume: Whenever we are left of a variable that is not the first argument of one of the righthand side predicates, we resume the clause of the righthand side predicate.

$$\frac{[A(\vec{\phi}) \to \ldots B(\vec{\xi}) \ldots, pos, \langle i,j\rangle, \vec{\rho}_A], \quad [B(\vec{\psi}) \to \vec{\Psi}, pos', \langle k-1, l\rangle, \vec{\rho}_B]}{[B(\vec{\psi}) \to \vec{\Psi}, pos, \langle k, 0\rangle, \vec{\rho}_B]},$$

where

- $\vec{\phi}(i)(j+1) = \vec{\xi}(k), k > 1$ (the next element is a variable that is the kth element in $\vec{\xi}$, i.e., the kth argument of B),
- $|\vec{\psi}(k-1)| = l$, and
- $\vec{\rho}_A(\vec{\xi}(h))$ (a range) and $\vec{\rho}_B(\vec{\psi})(h)$ (a sequence of ranges and terminals) are compatible for all $1 \leq h \leq k-1$.



$S(X) \to \langle \alpha \rangle(X)$
$\langle \alpha \rangle(LR) \to \langle adj, \alpha, \varepsilon \rangle(L,R)$
$\langle \beta \rangle(aLb, cRd) \to \langle adj, \beta, 2 \rangle(L,R)$
$\langle adj, \alpha, \varepsilon \rangle(L,R) \to \langle \beta \rangle(L,R)$
$\langle adj, \beta, 2 \rangle(L,R) \to \langle \beta \rangle(L,R)$
$\langle adj, \alpha, \varepsilon \rangle(\varepsilon, \varepsilon) \to \varepsilon$
$\langle adj, \beta, 2 \rangle(\varepsilon, \varepsilon) \to \varepsilon$

Successful items obtained from parsing $w = abcd$:

	dotted clause	*pos*	bindings $\vec{\rho}$	operation
1	$S(\bullet X) \to \langle \alpha \rangle(X)$	0	[?]	
2	$\langle \alpha \rangle(\bullet LR) \to \langle adj, \alpha, \varepsilon \rangle(L,R)$	0	[?, ?]	predict
3	$\langle adj, \alpha, \varepsilon \rangle(\bullet L, R) \to \langle \beta \rangle(L,R)$	0	[?, ?]	predict
4	$\langle \beta \rangle(\bullet aLb, cRd) \to \langle adj, \beta, 2 \rangle(L,R)$	0	[?, ?]	predict
5	$\langle \beta \rangle(a \bullet Lb, cRd) \to \langle adj, \beta, 2 \rangle(L,R)$	1	[?, ?]	scan
6	$\langle adj, \beta, 2 \rangle(\bullet, \varepsilon) \to \varepsilon$	1	[]	predict
7	$\langle \beta \rangle(aL \bullet b, cRd) \to \langle adj, \beta, 2 \rangle(L,R)$	1	[⟨1,1⟩, ?]	suspend
8	$\langle \beta \rangle(aLb \bullet, cRd) \to \langle adj, \beta, 2 \rangle(L,R)$	2	[⟨1,1⟩, ?]	scan
9	$\langle adj, \alpha, \varepsilon \rangle(L \bullet, R) \to \langle \beta \rangle(L,R)$	2	[⟨0,2⟩, ?]	suspend
10	$\langle \alpha \rangle(L \bullet R) \to \langle adj, \alpha, \varepsilon \rangle(L,R)$	2	[⟨0,2⟩, ?]	suspend
11	$\langle adj, \alpha, \varepsilon \rangle(L, \bullet R) \to \langle \beta \rangle(L,R)$	2	[⟨0,2⟩, ?]	resume
12	$\langle \beta \rangle(aLb, \bullet cRd) \to \langle adj, \beta, 2 \rangle(L,R)$	2	[⟨1,1⟩, ?]	resume
13	$\langle \beta \rangle(aLb, c \bullet Rd) \to \langle adj, \beta, 2 \rangle(L,R)$	3	[⟨1,1⟩, ?]	scan
14	$\langle adj, \beta, 2 \rangle(\varepsilon, \bullet) \to \varepsilon$	3	[]	resume
15	$\langle \beta \rangle(aLb, cR \bullet d) \to \langle adj, \beta, 2 \rangle(L,R)$	3	[⟨1,1⟩, ⟨3,3⟩]	suspend
16	$\langle \beta \rangle(aLb, cRd \bullet) \to \langle adj, \beta, 2 \rangle(L,R)$	3	[⟨1,1⟩, ⟨3,3⟩]	scan
17	$\langle adj, \alpha, \varepsilon \rangle(L, R \bullet) \to \langle \beta \rangle(L,R)$	4	[⟨0,2⟩, ⟨2,4⟩]	suspend
18	$\langle \alpha \rangle(LR \bullet) \to \langle adj, \alpha, \varepsilon \rangle(L,R)$	4	[⟨0,2⟩, ⟨2,4⟩]	suspend
19	$S(X \bullet) \to \langle \alpha \rangle(X)$	4	[⟨0,4⟩]	

Fig. 8. Source TAG, converted simple RCG, and a sample parsing trace

The goal items have the form $[S(\vec{\phi}) \to \vec{\Phi}, n, \langle 1, j\rangle, \psi]$ with $|\vec{\phi}(1)| = j$ (i.e., the dot is at the end of the lefthand side arguments).

With a dynamic programming interpretation, i.e., implemented as a chart parser, this algorithm can run in polynomial time [16].

Figure 8 shows a sample TAG, the corresponding RCG and a sample parse trace.

4.3. Parsing RCG. As mentioned before, our parsing architecture includes implementations of two parsing algorithms for the full class of RCG, namely of Boullier's algorithm [10] and the constraint-based parsing algorithm introduced in [15]. Both of them will be presented in the following in an incremental fashion, i.e., as extensions of simpler algorithms, in order to highlight the properties of each one.

For simplicity, we assume in the following without loss of generality that empty arguments (ε) occur only in clauses whose righthand-sides are empty[8].

The idea of top-down parsing is to instantiate the start predicate with the entire string and to recursively check if there is a way to reduce all righthand side predicates to ε.

Top-down parsing. The idea of top-down parsing is to instantiate the start predicate with the entire string and to recursively check if there is a way to reduce all righthand side predicates to ε.

Non-directional top-down parsing. The items have the form $[A, \phi, flag]$, where A is a predicate, ϕ is a range vector of dimension $dim(A)$ (containing the ranges that the arguments of A are instantiated with) and $flag \in \{c, p\}$ indicates if the item has been completed or predicted.

As an axiom, we predict S ranging over the entire input. Therefore, the **initialize** rule is as follows:

$$\frac{}{[S, (\langle 0, n\rangle), p]}.$$

The **predict** operation predicts new items for previously predicted items.

$$\frac{[A_0, \phi, p]}{[A_1, \phi_1, p] \dots [A_k, \phi_k, p]}.$$

Thereby, the following must hold:
There is a clause $c = A_0(\vec{x}_0) \to A_1(\vec{x}_1) \dots A_k(\vec{x}_k)$ with an instantiation ψ such that $\psi(c) = A(\phi) \to A_1(\phi_1) \dots A_k(\phi_k)$.

Since, unlike in standard top-down parsing for context-free grammar, we already start off with the entire string at initialization time, we need a way to propagate information about successful predicates. This is achieved by the p/c-flag, which is set by the scan and the complete operations.

The **scan** operation switches the flag on a item describing a predicted predicate to completed.

$$\frac{[A, \phi, p]}{[A, \phi, c]}$$

under the condition that there is a clause $c = A(\vec{x}) \to \varepsilon$ with an instantiation ψ such that $\psi(A(\vec{x})) = A(\phi)$.

The **complete** rule sets the flag on a completed lefthand side predicate to completed.

$$\frac{[A_0, \phi, p], [A_1, \phi_1, c] \dots [A_k, \phi_k, c]}{[A_0, \phi, c]}.$$

Thereby, the side conditions on the items in the antecendent of the rule are identical with the side conditions on all items of the predict rule.

Recognition is successful if there is a way to declare the start predicate completed. Consequently, the **goal** item is $[S, (\langle 0, n\rangle), c]$.

Directional top-down parsing. The above algorithm can be improved by evaluating righthand side predicates from left to right and stopping further evaluation once a predicate fails. This variant corresponds to the algorithm presented in [10].

For the directional top-down parsing algorithm, we need to distinguish between *passive items* and *active items*. Passive items have the same form and meaning as the items of the non-directional top-down parsing algorithm. Active items allow us to move a dot through the righthand side of a clause: $[A(\vec{x}) \to \Phi \bullet \Psi, \phi]$ where $A(\vec{x}) \to \Phi\Psi$ is a clause and ϕ is a range vector of dimension $j = \Upsilon(A(\vec{x}) \to \Phi\Psi)$ that gives an instantiation of the clause.

The axiom is the prediction of the start predicate ranging over the entire input. The **initialize** rule is the same as in the non-directional top-down case.

We have two predict operations. The first one, **predict-rule**, predicts active items with the dot on the left of the righthand side, for a given predicted passive item.

$$\frac{[A, \psi, p]}{[A(\vec{x}) \to \bullet\Psi, \phi]} \phi(A(\vec{x})) = A(\psi).$$

Predict-pred predicts a passive item for the predicate following the dot in an active item:

$$\frac{[A(\vec{x}) \to \Phi \bullet B(\vec{y})\Psi, \phi]}{[B, \psi, p]} \phi(B(\vec{y})) = B(\psi).$$

The **scan** operation is the same as in the non-directional case.

Complete moves the dot over a predicate in the righthand side of an active item if the corresponding passive item has been completed.

$$\frac{[B, \phi_B, c], [A(\vec{x}) \to \Phi \bullet B(\vec{y})\Psi, \phi]}{[A(\vec{x}) \to \Phi B(\vec{y}) \bullet \Psi, \phi]},$$

where $\phi(B(\vec{y})) = B(\phi_B)$.

Once the dot has reached the right end of a clause, we can **convert** the active item into a *completed* passive item:

$$\frac{[A(\vec{x}) \to \Phi\bullet, \phi]}{[A, \psi, c]} \phi(A(\vec{x})) = A(\psi).$$

[8] Any RCG can be easily transformed into an RCG satisfying this condition: Introduce a new unary predicate Eps with a clause $Eps(\varepsilon) \to \varepsilon$. Then, for every clause c with righthand-side not ε, replace every argument ε that occurs in c with a new variable X_ε and add the predicate $Eps(X_\varepsilon)$ to the righthand-side of c.

The **goal** item is again $[S,(\langle 0,n\rangle),c]$.

An obvious problem of this algorithm is that **predict-rule** has to compute all possible instantiations of A-clauses, given an instantiated A-predicate. Take for example the RCG for $\{a^{2^n} \mid n \geq 0\}$. If $w = aaaa$, starting from $[S,(\langle 0,4\rangle),p]$ **predict-rule** would predict (among others) all active items $[S(X_1Y_2) \rightarrow \bullet S(X_1)eq(X_1,Y_2),(\langle 0,r\rangle,\langle r,4\rangle)]$ with $r \in \{0,1,2,3,4\}$.

The computation of all these possible instantiations is very costly and will be avoided in the Earley algorithm that we present later on. In fact, the latter will use range constraint vector (instead of range vectors) and predict only one active item $[S(X_1Y_2) \rightarrow \bullet S(X_1)eq(X_1,Y_2),\langle\langle(r_1,r_2),(r_3,r_4)\rangle,\{0 = r_1,\ r_1 \leq r_2,\ r_2 = r_3,\ r_3 \leq r_4,\ 4 = r_4\}\rangle]$.

Bottom-up chart parsing. A CYK (Cocke, Younger, Kasami) style parser (non-directional bottom-up parsing) is the preliminary step to our Earley-style algorithm.

CYK parsing. The items have the form $[A,\phi]$ where A is a predicate and ϕ a range vector of dimension $dim(A)$.

$$\textbf{Scan}: \frac{}{[A,\phi]}.$$

Thereby, the following must hold:
There is a clause $c = A(\vec{x}) \rightarrow \varepsilon$ with an instantiation ψ such that $\psi(A(\vec{x})) = A(\phi)$.

$$\textbf{Complete}: \frac{[A_1,\phi_1]\ldots[A_k,\phi_k]}{[A,\phi]},$$

where $A(\phi) \rightarrow A_1(\phi_1)\ldots A_k(\phi_k)$ is an instantiated clause.

The **goal** item is $[S,(\langle 0,n\rangle)]$.

Directional bottom-up parsing. An obvious disadvantage of the basic CYK algorithm is that, in order to perform a *complete* step, all $A_1,\ldots,A_k$ in the righthand side must be checked for appropriate items. This leads to a lot of indices that need to be checked at the same time.

To avoid this, we can again move a dot through the righthand side of a clause. As in the case of the directional top-down algorithm, in addition to the items used above which we call the *passive* items now, we also need *active* items. In the active items, while traversing the righthand side of the clause, we keep a record of the positions already found for the left and right boundaries of variables and terminal occurrences. This is achieved by subsequently enriching the range constraint vector of the clause.

Active items have the form $[A(\vec{x}) \rightarrow \Phi \bullet \Psi,\langle\rho,C\rangle]$ with $A(\vec{x}) \rightarrow \Phi\Psi$ a clause, $\Phi\Psi \neq \varepsilon$, $\Upsilon(A(\vec{x} \rightarrow \Phi\Psi)) = j$ and $\langle\rho,C\rangle$ a range constraint vector of dimension j. We require that $\langle\rho,C\rangle$ be satisfiable.

Items that are distinguished from each other only by a bijection of the range variables are considered equivalent. I.e., if the application of a rule yields a new item such that an equivalent one has already been generated, this new one is not added to the set of partial results.

The **scan** rule is the same as in the basic algorithm. In addition, we have an **initialize** rule that introduces clauses with the dot on the left of the righthand side:

$$\frac{}{[A(\vec{x}) \rightarrow \bullet\Phi,\langle\rho,C\rangle]}$$

$A(\vec{x}) \rightarrow \Phi$ being a clause with range constraint vector $\langle\rho,C\rangle, \Phi \neq \varepsilon$.

The **complete** rule moves the dot over a predicate in the righthand side of an active item provided the corresponding passive item has been completed:

$$\frac{[A(\vec{x}) \rightarrow \Phi \bullet B(x_1\ldots y_1,\ldots,x_k\ldots y_k)\Psi,\langle\rho,C\rangle]}{[A(\vec{x}) \rightarrow \Phi B(x_1\ldots y_1,\ldots,x_k\ldots y_k) \bullet \Psi,\langle\rho,C'\rangle]},$$

where

$$C' = C \cup \{\phi_B(j).l = \rho(\Upsilon(x_j)).l,\ \phi_B(j).r = \rho(\Upsilon(y_j)).r \mid 1 \leq j \leq k\}.$$

Grammar for $\{a^{2^n} \mid n \geq 0\}$: $S(XY) \rightarrow S(X)eq(X,Y)$, $S(a_1) \rightarrow \varepsilon$, $eq(a_1X,a_2Y) \rightarrow eq(X,Y)$, $eq(a_1,a_2) \rightarrow \varepsilon$
Items generated for input $w = aa$ (the constraints $0 \leq r_1,r_2 \leq n$ for a range $\langle r_1,r_2\rangle$ are omitted):

	Item	operation
1.	$[S,(\langle 0,1\rangle)]$	scan $S(a_1) \rightarrow \varepsilon$
2.	$[S,(\langle 1,2\rangle)]$	scan $S(a_1) \rightarrow \varepsilon$
3.	$[eq,(\langle 0,1\rangle,\langle 0,1\rangle)]$	scan $eq(a_1,a_2) \rightarrow \varepsilon$
4.	$[eq,(\langle 0,1\rangle,\langle 1,2\rangle)]$	scan $eq(a_1,a_2) \rightarrow \varepsilon$
5.	$[eq,(\langle 1,2\rangle,\langle 0,1\rangle)]$	scan $eq(a_1,a_2) \rightarrow \varepsilon$
6.	$[eq,(\langle 1,2\rangle,\langle 1,2\rangle)]$	scan $eq(a_1,a_2) \rightarrow \varepsilon$
7.	$[S(XY) \rightarrow \bullet S(X)eq(X,Y),\{X.l \leq X.r, X.r = Y.l, Y.l \leq Y.r\}]$	initialize
8.	$[eq(a_1X,a_2Y) \rightarrow \bullet eq(X,Y),$ $\{a_1.l+1 = a_1.r, a_1.r = X.l, X.l \leq X.r, a_2.l+1 = a_2.r, a_2.r = Y.l, Y.l \leq Y.r\}]$	initialize
9.	$[S(XY) \rightarrow S(X) \bullet eq(X,Y),\langle\{\ldots,0 = X.l, 1 = X.r\}]$	complete 7. with 1.
10.	$[S(XY) \rightarrow S(X) \bullet eq(X,Y),\{\ldots,1 = X.l, 2 = X.r\}]$	complete 7. with 2.
11.	$[eq(a_1X,a_2Y) \rightarrow eq(X,Y)\bullet,\{\ldots,1 = X.l, 2 = X.r, 1 = Y.l, 2 = Y.r\}]$	complete 8. with 6.
12.	$[S(XY) \rightarrow S(X)eq(X,Y)\bullet,\{\ldots,0 = X.l, 1 = X.r, 1 = Y.l, 2 = Y.r\}]$	complete 9. with 4.
13.	$[eq,(\langle 0,2\rangle,\langle 0,2\rangle)]$	convert 11.
14.	$[S,(\langle 0,1\rangle,\langle 1,2\rangle)]$	convert 12.

Fig. 9. Trace of a sample CYK parse

Parsing trace for $w = aa$:

	Item	Rule
1	$[S, \langle(\langle r_1, r_2\rangle), \{0 = r_1, r_1 \leq r_2, 2 = r_2\}\rangle, p]$	initialize
2	$[S(XY) \rightarrow \bullet S(X) eq(X,Y), \{X.l \leq X.r, X.r = Y.l, Y.l \leq Y.r, 0 = X.l, 2 = Y.r\}]$	predict-rule from 1
3	$[S, \langle(\langle r_1, r_2\rangle), \{0 = r_1, r_1 \leq r_2\}\rangle, p]$	predict-pred from 2
4	$[S, (\langle 0, 1\rangle), c]$	scan from 3
5	$[S(XY) \rightarrow \bullet S(X) eq(X,Y), \{X.l \leq X.r, X.r = Y.l, Y.l \leq Y.r, 0 = X.l, \}]$	predict-rule from 3
6	$[S(XY) \rightarrow S(X) \bullet eq(X,Y), \{\ldots, 0 = X.l, 2 = Y.r, 1 = X.r\}]$	complete 2 with 4
7	$[S(XY) \rightarrow S(X) \bullet eq(X,Y), \{X.l \leq X.r, X.r = Y.l, Y.l \leq Y.r, 0 = X.l, 1 = X.r\}]$	complete 5 with 4
8	$[eq, \langle(\langle r_1, r_2\rangle, \langle r_3, r_4\rangle), \{r_1 \leq r_2, r_2 = r_3, r_3 \leq r_4, 0 = r_1, 2 = r_4, 1 = r_2\}\rangle]$	predict-pred from 6
9	$[eq(a_1X, a_2Y) \rightarrow \bullet eq(X,Y), \{a_1.l + 1 = a_1.r, a_1.r = X.l, X.l \leq X.r,$ $a_2.l + 1 = a_2.r, a_2.r = Y.l, Y.l \leq Y.r, X.r = a_2.l, 0 = a_1.l, 1 = X.r, 2 = Y.r\}]$	predict-rule from 8
	...	
10	$[eq, (\langle 0, 1\rangle, \langle 1, 2\rangle), c]$	scan 8
11	$[S(XY) \rightarrow S(X) eq(X,Y) \bullet, \{\ldots, 0 = X.l, 2 = Y.r, 1 = X.r, 1 = Y.l\}]$	complete 6 with 10
12	$[S, (\langle 0, 2\rangle), c]$	convert 11

Fig. 10. Trace of a sample Earley parse

Note that the conditions on the items require the new constraint set for ρ to be satisfiable.

Convert turns an active item with the dot at the end of the righthand-side into a completed passive item:

$$\frac{[A(\vec{x}) \rightarrow \Psi \bullet, \langle \rho, C\rangle]}{[A, \phi]},$$

where there is an instantiation ψ of $A(\vec{x}) \rightarrow \Psi$ that satisfies $\langle \rho, C\rangle$ such that $\psi(A(\vec{x})) = A(\phi)$.

The **goal** item is $[S, (\langle 0, n\rangle)]$.

A sample parse trace is shown in Fig. 9. For the sake of readability, instead of the range boundary variables, we use $X.l$ and $X.r$ respectively for the left and right range boundary of the range associated with X.

The Earley algorithm.

Deduction rules. We now add a prediction operation to the CYK algorithm with active items which leads to an Earley-style algorithm. As we have seen before, the passive items are enriched with an additional flag that can have values p or c depending on whether the item is only predicted or already completed. Furthermore, they contain range constraint vectors since when predicting a category, the left and right boundaries of its arguments might not be known.

Passive items either have the form $[A, \langle \rho, C\rangle, p]$ for a predicted item, where $\langle \rho, C\rangle$ is a range constraint vector of dimension $dim(A)$, or the form $[A, \phi, c]$ for completed items where ϕ is a range vector of dimension $dim(A)$. The active items are the same as in the CYK case.

The axiom is the prediction of an S ranging over the entire input, i.e., the **initialize** rule is as follows:

$$\frac{}{[S, \langle(\langle r_1, r_2\rangle), \{0 = r_1, n = r_2\}\rangle, p]}.$$

We have two predict operations. The first one, **predict-rule**, predicts active items with the dot on the left of the righthand side, for a given predicted passive item:

$$\frac{[A, \langle \rho, C\rangle, p]}{[A(x_1 \ldots y_1, \ldots, x_k \ldots y_k) \rightarrow \bullet \Psi, \langle \rho', C'\rangle]},$$

where $\langle \rho', C'\rangle$ is obtained from the range constraint vector of the clause $A(x_1 \ldots y_1, \ldots, x_k \ldots y_k) \rightarrow \Psi$ by taking all constraints from C, mapping all $\rho(i).l$ to $\rho'(\Upsilon(x_i)).l$ and all $\rho(i).r$ to $\rho'(\Upsilon(y_i)).r$, and then adding the resulting constraints to the range constraint vector of the clause.

The second predict operation, **predict-pred**, predicts a passive item for the predicate following the dot in an active item:

$$\frac{[A(\ldots) \rightarrow \Phi \bullet B(x_1 \ldots y_1, \ldots, x_k \ldots y_k)\Psi, \langle \rho, C\rangle]}{[B, \langle \rho', C'\rangle, p]},$$

where $\rho'(i).l = \rho(\Upsilon(x_i)).l$, $\rho'(i).r = \rho(\Upsilon(y_i)).r$ for all $1 \leq i \leq k$ and $C' = \{c \mid c \in C, c$ contains only range variables from $\rho'\}$.

The **scan** rule can be applied if a predicted predicate can be derived by an ε-clause:

$$\frac{[A, \langle \rho, C\rangle, p]}{[A, \phi, c]},$$

where there is a clause $A(\vec{x}) \rightarrow \varepsilon$ with a possible instantiation ψ that satisfies $\langle \rho, C\rangle$ such that $\psi(A(\vec{x})) = A(\phi)$.

Finally, deduction rules for **complete** and **convert** are the ones from the CYK algorithm with active items except that we add flags c to the passive items occurring in these rules.

Again, the **goal** item is $[S, (\langle 0, n\rangle), c]$.

To understand how this algorithm works, consider the example in Fig. 10, with the RCG and the input word from Fig. 9.

Note that the algorithm shows a great similarity to the directional top-down algorithm. The crucial difference is that while in the top-down algorithm, we are using range vectors to record the variable bindings, in the Earley-style algorithm, we use range constraint vectors. Due to the fact that range constraint vectors allow us to leave range boundaries unspecified, we can compute the value of range boundaries in a more incremental fashion since we do not have to guess all values of all boundary variables of a clause at once as in the top-down algorithm. This becomes particularly clear when comparing the **complete** rules of the non-directional top-down algorithm and the Earley-style algorithm. In the former, we check the

compatibility of the range vector of the completed item with the range vector of the item which is to be completed as a side condition. In the latter however, we add the information contributed by the range vector of the completed item dynamically to the range constraint vector of the item to be completed.

The directional bottom-up parser from the section titled "Bottom-up chart parsing", in contrast to the Earley algorithm, lacks the top-down predictions. However, it uses the same technique of dynamic updating of a set of constraints on range boundaries, therefore the active items are the same for the two algorithms.

Soundness and completness. It is easy to see that the Earley-style algorithm is both sound and complete. More precisely, if a completed item is generated, then the corresponding predicate can be derived: $[A, \psi, c] \Rightarrow A(\psi)$. Furthermore, if we can derive a constituent $A(\psi)$, we also generate the corresponding item. Let Γ be a string of instantiated predicates. Then

$$S(\langle 0,n\rangle) \stackrel{*}{\Rightarrow}_l A(\psi)\Gamma \stackrel{*}{\Rightarrow}_l \Gamma \text{ iff } [A,\psi,c],$$

where $\stackrel{*}{\Rightarrow}_l$ signifies "leftmost derivation".

In particular, $[S,(\langle 0,n\rangle),c]$ iff $S(\langle 0,n\rangle) \stackrel{*}{\Rightarrow} \varepsilon$.

Obtaining a parse forest. So far, we have described recognizers, not parsers. The way to obtain a parse forest from the item set resulting from the Earley recognizer with range boundary constraints is rather obvious. Whenever a convert is done, a fully instantiated clause has been found. By collecting these clauses, we obtain a compact representation of our parse forest[9]. Starting from an S predicate ranging over the entire input and following the clauses for the instantiated predicates in the righthand sides, we can read off the single parse trees from this forest.

4.4. Computing semantics. The parsing architecture introduced here has been extended to support the syntax/semantics interface of Gardent and Kallmeyer [17]. The underlying idea of this interface is to associate each tree with flat semantic formulas. The arguments of these formulas are unification variables co-indexed with features labelling the nodes of the syntactic tree. During derivation, trees are combined via adjunction and/or substitution, each triggering the unifications of the feature structures labelling specific nodes. As a result of these unifications, the arguments of the semantic formulas associated with the trees involved in the derivation get unified.

5. Implementing TAG parsing using RCG: TuLiPA

The transformation algorithm introduced in Sec. 3, along with the RCG parsing algorithms presented in Sec. 4 have been implemented within the TuLiPA system. Let us give some more details about its architecture and features.

TuLiPA is a *modular* parsing architecture, which takes as an input a grammar (associated with lexical and syntactic resources)[10] in an XML format. The formalisms currently supported are TAG, TT-MCTAG, RCG, and, as a special case of the preceding formalisms, Context-Free Grammar. The DTD specifications of the XML format expected by TuLiPA are given in the TuLiPA website[11]. Since TuLiPA expects no specific format, only XML-based grammars, it does not imply any constraint about how the grammar has been obtained. It could have been hand-crafted, or compiled from an abstract representation, using for instance the eXtensible MetaGrammar language [18][12].

5.1. Preprocessings. First, the parser loads the grammar and its associated lexica into memory. These data remain in memory as long as the parser is not closed, allowing to parse several sentences without any reloading (nonetheless, if the resources have been modified, reloading can be enforced). Secondly, it processes the input sentence in a classical way: *tokenization*, and *part-of-speech labelling*. TuLiPA includes a basic tokenizer and part-of-speech tagger, or it can use an external POS-tagger (currently only the TreeTagger[13] developed at Universität Stuttgart is supported).

Then TuLiPA retrieves the subgrammar selected by the labelled input sentence, and, at the same time, performs *tree anchoring*. The grammar that is fed to the parsing architecture is supposed to be made of tree templates, which are basically trees with a distinguished leaf node (sometimes labelled with a diamond in the TAG litterature). This node is where the lexical item is to be anchored (if the tree is lexicalized). The anchoring relies on both the concept of *tree family* (tree are gathered according to the subcategorization frame they encode), and the unification of syntactic features labelling both the entries of the lexicon, and the tree templates (the features labelling the templates are sometimes called *hypertags* [20]).

Prior to RCG-convertion, the size of the subgrammar is reduced by applying *lexical disambiguation* [21]. Basically, we use automata-based techniques to precompute sets of compatible grammatical structures (i.e., compatible anchored trees for TAG, compatible anchored tree tuples for TT-MCTAG). For instance, consider the sentence *John eats a cake*. If the verb *to eat* is associated to both the tree families *transitive* and *intransitive* in the lexicon, the trees of the two families will be

[9]Note that, strictly speaking, this structure is not the parse forest as it contains some clause instantiations that are not part of the actual parse forest. This is not a problem for these useless instantiations are ignored when reading the parses starting from the instantiated S predicates.

[10]Following the XTAG parsing architecture [19], the linguistic resources are splitted in 3 lexica: the tree templates, the lemmas and the morphological items.

[11]See http://sourcesup.cru.fr/tulipa

[12]Note that the TT-MCTAG for German developped within the Emmy Noether Project at Universität Tübingen (see http://www.sfs.uni-tuebingen.de/emmy/res-en.html) has been developped using eXtensible MetaGrammar (http://sourcesup.cru.fr/xmg).

[13]See http://http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/

loaded by the parser. From the presence of two noun phrases in the sentence, the parser should be able to consider only the transitive family.

To perform this disambiguation, we build an automaton listing the potential families to be selected for each token of the input string. These families are transitions of the automaton. Its states are *disambiguation features*. In the TAG case, we automatically label trees with disambiguation features indicating whether a tree needs some syntactic material (i.e., it has substitution nodes with given labels), or it brings some material of a specific type (via the label of its root node).

We then add an initial state to this automaton with an empty set of features. From this state, there are transitions for all potential tree family for the first token. These transitions lead to states whose sets of disambiguation features is computed using the features labelling the tree of the considered family (needs are negative features, resources are positive features, if the number of positive and negative features for a given feature f are the same, f is removed from the set of disambiguation features). We go on by adding transitions for the families of the second token, which lead to new states, and so on[14]. In the end (all potential families of all tokens have been processed), the automaton is traversed. All paths leading to the final state, whose sets of features only consits of $+S$ (if we are parsing a sentence), are kept. The families in these paths are the subgrammar fed to the RCG-transformation module. As an illustration, see Fig. 11.

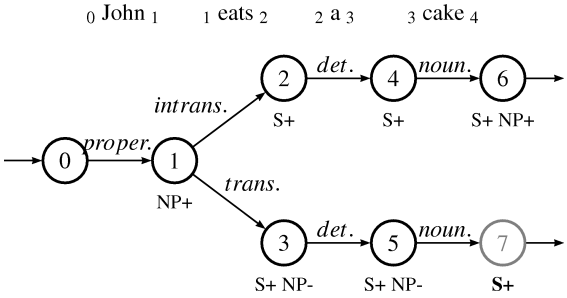


Fig. 11. Lexical disambiguation

Once the subgrammar has been selected from the input string, it is fed to the RCG-converter, which implements the algorithm of Sec. 3.

5.2. Core of the parser. From the RCG computed for the input string, we can choose between the implementations of several RCG parsing algorithms, all of which have been presented in the previous section. Note that all algorithms can be called directly, i.e., apart from TAG parsing and TT-MCTAG parsing, direct parsing of RCG, simple RCG (resp. LCFRS), and CFG is also supported.

The result of parsing is an RCG derivation forest, encapsulating all RCG derivations. This forest corresponds to an AND-OR graph, which states which grammatical rules (i.e., RCG clauses) have been used during a parse, and how (in which context). Note that OR nodes appear when there is grammatical ambiguity.

5.3. Postprocessings.

From RCG-derivation forest to TAG-derivation forest. Once the RCG-derivation forest has been computed, we still need to process it to extract the underlying TAG parse forest. This extraction is done in a single traversal of the forest by interpreting the LHS predicate names of the clauses associated with the nodes of the parse forest. As an example, consider the fragment of a parse forest presented in Fig. 12. This figure shows 3 clauses encoding a root adjunction of the tree β_v (argument tree of the tuple anchored by *versprechen*) to the tree α_{rep} (head tree of the tuple anchored by *zu reparieren*).

$$\langle \alpha_{rep}, \emptyset \rangle (es\ der\ Mech\ zu\ rep\ versp) \rightarrow$$
$$\langle adj, \alpha_{rep}, \varepsilon, \{\beta_{acc}\} \rangle (es\ der\ Mech, versp)$$

$$\langle adj, \alpha_{rep}, \varepsilon, \{\beta_{acc}\} \rangle (es\ der\ Mech, versp) \rightarrow$$
$$\langle \beta_v, \{\beta_{acc}\} \rangle (es\ der\ Mech, versp)$$

$$\langle \beta_v, \{\beta_{acc}\} \rangle (es\ der\ Mech, versp) \rightarrow$$
$$\langle adj, \beta_v, \varepsilon, \{\beta_{acc}, \beta_{nom}\} \rangle (es\ der\ Mech., \varepsilon)$$

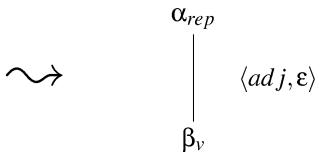


Fig. 12. Retrieving TT-MCTAG derivations from a parse forest for the RCG in Fig. 7

Unpacking the TAG derivation forest and computing semantic representations. The TAG derivation forest is then unpacked in order to retrieve every single valid TAG parse. At the same time, we perform the unifications related to the underlying TAG adjunctions and substitutions. In other terms, during parsing, we did not take into account the feature structure labelling the TAG trees (we only used them for anchoring). It seems reasonable to perform these TAG-based unifications only as a postprocessing, for it increases the complexity of both RCG-transformation (control of the adjunctions) and RCG parsing (RCG clauses have to be enriched with features structures), while RCG forest usually do not contain much spurious parse ambiguities.

Furthermore, while we compute these unifications, we get semantic representations for free, provided the TAG trees have been extended with a syntax/semantic interface *à la* Gardent and Kallmeyer [17]. Basically, each TAG tree is extended with an underspecified semantic formula that is inspired by an extension of First-Order Logic called Predicate Logic Unplugged [22]. The arguments of the predicates of these semantic formulas are shared with specific features labelling the nodes of the syntactic TAG tree. During TAG derivation, the unifications of these feature structures lead to an update

[14]Note that during the construction of the automaton, we reduces it by merging states having similar sets of disambiguation features.

of the arguments of the semantic formulas. As an illustration, consider Fig. 13, which shows how to extends the trees associated with *John*, *loves* and *Mary*, in order to compute the semantic representation of *John loves Mary*[15].

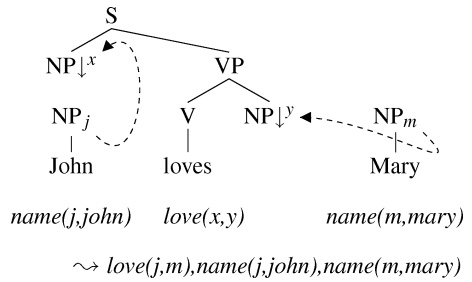


Fig. 13. Semantic calculus in Feature-Based TAG

5.4. TuLiPA's ouput. As an output, TuLiPA either dumps an XML file containing the parse trees, or displays the derivation trees in a graphical user interface (see Fig. 14).

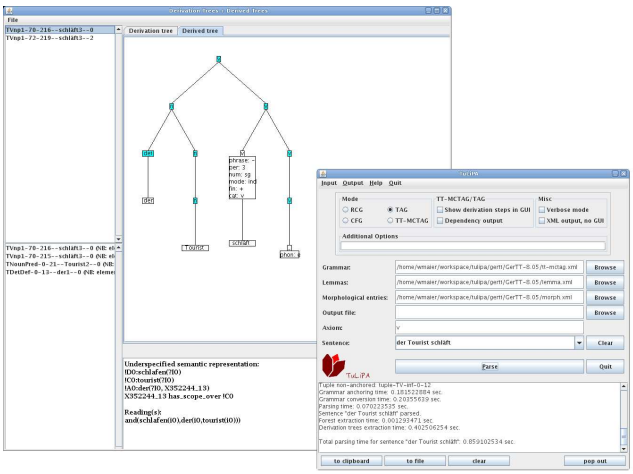


Fig. 14. TuLiPA's GUI

Besides derivation trees (and the corresponding derived trees), TuLiPA also returns semantic representations (underspecified or fully specified ones, the latter being computed by the Utool[16] system integrated in TuLiPA) or dependency views of the derivation trees (using the Dtool software)[17]. Furthermore, TuLiPA includes a robust mode computing partial parses, and displaying feature mismatches in the graphical user interface.

6. Conclusions and future work

In this paper, we introduced a parsing environment using RCG as a pivot formalism to parse extensions of TAG, such as TT-MCTAG. The motivation to this lies in RCG's formal (e.g. expressivity) and computational (e.g. polynomial parsing time) properties. In particular, it is possible to encode TAG and extensions of TAG into equivalent RCG, for which there exists efficient parsing algorithms. The ideas introduced in this paper have led to the implementation of TuLiPA, a parsing environment relying on a modular architecture performing several tasks: TAG to RCG convertion, RCG parsing, and interpretation of the RCG derivation.

Future work will include experiments with off-line conversion of TT-MCTAG and generalization of branching clauses to reduce the size of the RCG and thus to improve (RCG) parsing time.

[15] More complex examples of this semantic calculus are given in [17].

[16] See http://www.coli.uni-saarland.de/projects/chorus/utool/, with courtesy of Alexander Koller.

[17] With courtesy of Marco Kuhlmann.

REFERENCES

[1] T. Lichte, "An MCTAG with tuples for coherent constructions in German", *Proc. 12th Conf. on Formal Grammar 2007* 1, 1–12 (2007).

[2] A.K. Joshi and Y. Schabes, "Tree-adjoning grammars", *Handbook of Formal Languages* 1, 69–123 (1997).

[3] S. Gorn, "Explicit definitions and linguistic dominoes", *Proc. Conf. held at University of Western Ontario* and *Systems and Computer Science* 1, 77–115 (1967).

[4] D. Weir, "Characterizing midly-context sensitive grammar formalisms", *PhD Thesis*, University of Pennsylvania, Pennsylvania, 1988.

[5] L. Kallmeyer, "Tree-local multicomponent tree adjoining grammars with shared nodes", *Computational Linguistics* 31 (2), 187–225 (2005).

[6] T. Lichte and L. Kallmeyer, "Factorizing complementation in a TT-MCTAG for German", *Proc. The Ninth Int. Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+9)* 1, 1–8 (2008).

[7] L. Kallmeyer and Y. Parmentier, "On the relation between multicomponent tree adjoining grammars with tree tuples (TT-MCTAG) and range concatenation grammars (RCG)", *Proc. 2nd Int. Conf. on Language and Automata Theory and Applications* 5196, 263–274 (2008).

[8] L. Kallmeyer and G. Satta, "A polynomial-time parsing algorithm for TT-MCTAG", *Proc. 47th Conf. Association for Computational Linguistics and the 4th Int. Joint Conf. on Natural Language Processing of the Asian Federation of NLP* 1, 994–1002 (2009).

[9] P. Boullier, "On TAG parsing", *Proc. TALN 99, 6e Conf. Annuelle sur le Traitement Automatique des Langues Naturelles* 1, 75–84 (1999).

[10] P. Boullier, "Range concatenation grammars", *Proc. Sixth Int. Workshop on Parsing Technologies* 1, 53–64 (2000).

[11] E. Bertsch and M.-J. Nederhof, "On the complexity of some extensions of RCG parsing", *Proc. Seventh Int. Workshop on Parsing Technologies* 1, 66–77 (2001).

[12] P. Boullier, "Proposal for a natural language processing syntactic backbone", *INRIA Research Report* 3342, CD-ROM (1998).

[13] P. Boullier, "On TAG and multi-component TAG parsing", *INRIA Research Report* 3668, ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-3668.pdf (1999).

[14] L. Kallmeyer and W. Maier, "An incremental Earley parser for simple range concatenation grammar", *Proc. 11th Int. Conf. on Parsing Technologies* 1, 61–64 (2009).

[15] L. Kallmeyer, W. Maier, and Y. Parmentier, "An Earley parsing algorithm for range concatenation grammar", *Proc. Short Papers of the 47th Conf. Association for Computational Linguistics and the 4th Int. Joint Conf. on Natural Language Processing of the Asian Federation of Natural Language Processing* 1, 9–12 (2009).

[16] É. Villemonte de La Clergerie, "Parsing mildly context-sensitive languages with thread automata", *Proc. 19th Int. Conf. on Computational Linguistics* 1, 1–7 (2002).

[17] C. Gardent and L. Kallmeyer, "Semantic construction in FTAG", *Proc. 10th Int. Conf. European Chapter of the Association for Computational Linguistics* 1, 123–130 (2003).

[18] D. Duchier, J. Le Roux, and Y. Parmentier, "An NLP application with a multi-paradigm architecture", *Proc. 2nd Mozart-Oz Conference* 1, 175–187 (2004).

[19] The XTAG research group, "A lexicalized tree adjoining grammar for English", *Institute for Research in Cognitive Science, Research Report*, University of Pennsylvania, Pennsylvania, 2001.

[20] A. Kinyon, "Hypertags", *Proc. 18th Int. Conf. on Computational Linguistics* 1, 446–452 (2000).

[21] G. Bonfante, B. Guillaume, and G. Perrier, "Polarization and abstraction of grammatical formalisms as methods for lexical disambiguation", *Proc. 20th Int. Conf. on Computational Linguistics* 1, 303–309 (2004).

[22] J. Bos, "Predicate logic unplugged", *Proc. Tenth Amsterdam Colloqium* 1, 133–143 (1995).