# Polynomial Pregroup Grammars parse Context Sensitive Languages

Anne Preller

# Polynomial Pregroup Grammars parse Context Sensitive Languages

Anne Preller[*]
*Lirmm, 161, rue Ada, Montpellier, France*
preller@lirmm.fr

**Abstract**

Pregroup grammars with a possibly infinite number of lexical entries are *polynomial* if the length of type assignments for sentences is a polynomial in the number of words. Polynomial pregroup grammars are shown to generate the standard mildly context sensitive formal languages as well as some context sensitive natural language fragments of Dutch, Swiss-German or Old Georgian. A polynomial recognition and parsing algorithm handles the various grammars uniformly. It also computes a planar graph for the semantic cross-serial dependencies in the case of natural languages.

*keywords*: type logical grammar, pregroup grammar, proof graph, complement control, cross-serial dependency, mildly context sensitive language, Dutch subordinate clause, Swiss-German subordinate clause, Old Georgian noun phrase, incremental dependency parsing algorithm.

## 1   Introduction

The Pregroup Calculus was introduced in (Lambek 199) [11] as a simplification of the earlier Syntactic Calculus in (Lambek 1958) [10]. According to (Buszkowski 2001)[3], a pregroup grammar consists of a finite set of basic types and a dictionary (or lexicon) containing a finite number of words, each listed with a finite number of types from the Pregroup Calculus. These finite grammars are proved in [*ibidem*] to be weakly equivalent to context free grammars.

Both (Francez *et al.* 2007) [5] and (Stabler 2008) [24] extend pregroup grammars to mildly context-sensitive formal languages by adding new rules and/or constraints to the Pregroup Calculus. (Lambek 2008-1) [12] discusses the Dutch subordinate clause and remarks that a law of commutativity would solve the problem, but dismisses it as 'not allowed'.

The approach taken here is based on the belief that the computational efficiency and the semantic expressivity of pregroup grammars is based on the planar graphs representing derivations of the Pregroup Calculus. To keep them

---

1

intact, the definition of a lexicon is relaxed allowing an infinite number of types per word. The only restriction is that some polynomial in $l$ bounds the length of the concatenated type $T_1 \ldots T_l$ associated to $w_1 \ldots w_l$ for every type assignment $w_i : T_i$, $1 \le i \le l$ with a reduction to the sentence type. In the case of context free languages or the standard formal mildly context sensitive languages it is of degree 1, for Dutch or Swiss-German subordinate clauses it is of degree 2. The pregroup grammar generating (Michaelis, Kracht) [9]'s version of Old Georgian noun phrases is polynomial of degree 2. The description of the same noun phrases in (Bhatt and Joshi) [1] can be handled by a polynomial pregroup grammar of degree 1.

The natural language dictionaries presented here have entries that are triples formed by a word, a type and a meaning expression. The meaning of a sentence is computed from the chosen meanings of the words by substitution. The dictionaries are compositional in the sense of (Kracht 2007) [7]. Moreover, a linear parsing and tagging algorithm generates the (cross-serial) semantic dependencies.

Section 2 recalls the Pregroup Calculus and the geometrical structure of reductions, followed by an example how to compute the meaning of a sentence involving complement control in English. Section 3 introduces polynomial pregroup grammars for formal and natural context sensitive languages. Finally, Section 4 presents the parsing algorithm based on the geometrical structure of reductions, illustrated by an example of Dutch subordinate clauses.

## 2 Pregroup Calculus, Proof Graphs and Meaning

### 2.1 Pregroup Calculus and Reductions

The set of pregroup types $P(\mathcal{B})$ *generated by a partially ordered set* $\mathcal{B} = \langle B, \le \rangle$ is the free monoid generated by the set of *simple types*

$$\mathcal{S}(B) = \left\{ a^{(z)} : a \in B, z \in \mathbb{Z} \right\} .$$

The notation $a^{(z)}$ designates the ordered pair formed by the element $a$ of $B$ and the integer $z$. Elements $T \in P(\mathcal{B})$ are called *types*. In an equality $T = t_1 \ldots t_n$, it is always understood that the lower case $t_i$'s are simple types. In the case where $n = 0$, the string $t_1 \ldots t_n$ is empty, denoted $1$. It is the unit for the binary operation of concatenation in the free monoid. A *basic type* is a simple type of the form $a^{(0)}$. With a convenient lack of precision, $a$ and $a^{(0)}$ are identified and the elements of $B$ are referred to as basic types. The *left adjoint* and the *right adjoint* of a simple type $t = a^{(z)}$ are defined as

$$\begin{aligned} t^\ell &= (a^{(z)})^\ell = (a^{(z-1)}) &&\text{left adjoint} \\ t^r &= (a^{(z)})^r = (a^{(z+1)}) &&\text{right adjoint} . \end{aligned}$$

The binary *derivability relation* on types, denoted $\rightarrow$, is the smallest transitive relation containing $1 \rightarrow 1$ satisfying

$$
\begin{array}{lll}
\text{(Induced step)} & Sa^{(z)}T \rightarrow Sb^{(z)}T & \\
\text{(Generalized contraction)} & Sa^{(z)}b^{(z+1)}T \rightarrow ST & \quad (1) \\
\text{(Generalized expansion)} & ST \rightarrow Sa^{(z+1)}b^{(z)}T &
\end{array}
$$

where either $z$ is even and $a \leq b$ or $z$ is odd and $b \leq a$.

Note that the derivability relation $\rightarrow$ coincides with the partial order $\leq$ on the set of basic types. It is a partial preorder on types, but not an order, because it is not antisymmetric. Indeed, $a \rightarrow a(a^r a)$ by generalized expansion and $(aa^r)a \rightarrow a$ by generalized contraction, see (Buszkowski 2001) [4].

**Definition 1** (Dictionaries). *Let $\Sigma$ be a non-empty set. A pregroup dictionary for $\Sigma$ based on $\mathcal{B}$ is a map $\mathcal{D}$ defined on $\Sigma$ with values in the set of subsets of $\mathcal{P}(\mathcal{B})$. A type assignment of $w_1 \ldots w_n$ is a sequence of types $T_1, \ldots, T_n$ for which $T_i \in \mathcal{D}(w_i), 1 \leq i \leq n$. A lexical entry $w : T$ of $\mathcal{D}$ is an ordered pair $w \in \Sigma$ and $T \in \mathcal{D}(w)$. A dictionary is* discrete *if it is based on a discrete set, i.e. a set ordered by equality.*

*A pregroup grammar $\mathcal{G} = \langle \mathcal{D}, s \rangle$ for $\Sigma$ based on $\mathcal{B}$ consists of a pregroup dictionary $\mathcal{D}$ based on $\mathcal{B}$ and a distinguished basic type $s \in \mathcal{B}$. The* language of $\mathcal{G}$ *is the following subset of $\Sigma^*$*

$$\mathcal{L}_{\mathcal{G}} = \{w_1 \ldots w_n \in \Sigma^* : T_1 \ldots T_n \rightarrow s \text{ for some } T_i \in \mathcal{D}(w_i), 1 \leq i \leq n\}.$$

By definition, $T \rightarrow T'$ if and only if there is a sequence of types $T_1, \ldots, T_n$ such that $T_1 = T$, $T_n = T'$ and $T_i \rightarrow T_{i+1}$ is $1 \rightarrow 1$ or an instance of the pairs in (1) for $1 \leq i < n$. The derivations of the Pregroup Calculus can be characterized geometrically by proof graphs that have underlinks, overlinks and vertical links, see (Preller, Lambek) [17]. Underlinks are edges between simple types in the upper line (the antecedent) and stand for generalized contractions. Overlinks are edges between simple types in the lower line (the conclusion) and represent generalized expansions. Vertical links are edges between a simple type in the upper line and a simple type in the lower line. They code induced steps.

$$
\begin{array}{c}
\underline{a^\ell a}\, a^\ell \, \underline{a\, \underline{a^\ell a}\, a^r}\, a\boldsymbol{s} \\[18pt]
\overline{c^r\, c}\, a^\ell\, \overline{a\, \overline{b^r\, b}}\, \boldsymbol{s}\, \overline{c\, c^\ell}
\end{array}
\qquad (2)
$$

To check grammaticality of a string of words, only derivations without instances of Generalized Expansion are to be considered, see (Lambek 1999) [11]. The corresponding proof graphs are called *reductions*. If the set of basic types is discretely ordered the lower line and the vertical links can be omitted, as both are determined by the unlinked simple types in the upper line. Moreover, for a fixed type $T$, a reduction is determined by the position numbers of the

3

links. This representation of a reduction as a set of unordered pairs of position numbers assures an easy formulation of the parsing algorithm in Section 4.

$$a^\ell\, \underline{a\,\underline{a^\ell a}\,a^r}\,a\,\boldsymbol{s} \quad\longrightarrow\quad \boldsymbol{s} \qquad\qquad 1\ \underline{2\ \underline{3\ 4}\ 5}\ 6\ \ 7 \qquad \{1,6\}\ \{2,5\}\ \{3,4\}$$

$$a^\ell a\, a^\ell\, \underline{a a^r}\, a\,\boldsymbol{s} \quad\longrightarrow\quad \boldsymbol{s} \qquad\qquad \underline{1\ 2}\ \underline{3\ \underline{4\ 5}\ 6}\ \ 7 \qquad \{1,2\}\ \{3,6\}\ \{4,5\}$$

$$\tag{3}$$

Note that there may be more than one reduction between two given types. The next subsection gives an example how a reduction to the sentence type assembles the meaning of the words into a meaning of the sentence.

**Definition 2** (Reduction). *Let $1 \leq i_1 < \cdots < i_k \leq n$, $T = t_1 \ldots t_n$ and $T' = t_{i_1} \ldots t_{i_k}$ be given. A reduction $\mathcal{R}$ from $T$ to $T'$, in symbols $\mathcal{R} : T \Rightarrow T'$, is a non-directed graph $\mathcal{R} = \langle \{1, \ldots, n\}, R\rangle$ such that*

1) *for every $i \in \{1, \ldots, n\} - \{i_1, \ldots, i_k\}$ there is exactly one $j \in \{1, \ldots, n\}$ such that $\{i, j\} \in R$*
2) *if $\{i, j\} \in R$ then $i \neq j$, $i \notin \{i_1, \ldots, i_k\}$, $j \notin \{i_1, \ldots, i_k\}$*
3) *if $\{i, j\} \in R$ and $i < l < j$ then $\{l, m\} \in R$ for some $m$ with $i < m < j$*
4) *$t_i t_j \to 1$ for $i < j$ and $\{i, j\} \in R$*

*The vertices in $\{1, \ldots, n\}$ are called* positions *and the edges in $R$* underlinks.

Follow a few properties that intervene repeatedly in the proofs of Section 3.

**Definition 3** (Simple type occurrence). *An arbitrary type $T$ is said to be in $\mathcal{D}$ if $w : T$ is a lexical entry for some word $w$. A simple type $t$ occurs in $T = t_1 \ldots t_n$ if $t = t_i$ for some $i \leq n$. It occurs in $\mathcal{D}$, if it occurs in the type of some lexical entry of $\mathcal{D}$.*

For example, $a^\ell$ and $\boldsymbol{s}$ occur in $\boldsymbol{s}a^\ell$, but not the basic type $a$.

**Definition 4** (Modest type). *A type is modest, if all simple types occurring in it are basic, or right or left adjoints of basic types, but no basic type occurs with both its right and left adjoint. A dictionary is modest if it is based on a discrete set and every type obtained by concatenating types in the dictionary is modest.*

*A type $T = t_1 \ldots t_n$ is irreducible if $t_i t_{i+1} \not\to 1$ for $1 \leq i < n$. A type $T'$ is an irreducible form of $T$ if $T'$ is irreducible and $T \to T'$.*

For example, if $a$ and $b$ are basic then $aa^\ell b^r b$ is modest and irreducible. Every simple type is irreducible. The type constituting the top line of (3) is not modest.

The proofs of lemmas 2 - 6 below are straight forward. Lemma 1 is a special case of Lemma 4.5 in (Preller 2007-2) [19].

**Lemma 1** (Uniqueness). *Every modest type has a unique irreducible form and a unique reduction to its irreducible form.*

4

The property does not hold in general. Indeed, $ba^\ell aa^r a$ has the two irreducible forms $b$ and $ba^r a$, whereas (3) gives a type with two distinct reductions to the same irreducible form.

**Lemma 2.** *Let $T'$ be the irreducible form of a modest type $T = t_1 \ldots t_m$ and suppose that $t_i \ldots t_j \to 1$ for some $1 \le i < j \le m$. Then $t_1 \ldots t_{i-1} t_{j+1} \ldots t_m \to T'$.*

Again, the property above does not hold in general. For example, $ba^\ell aa^r a \to b$, $a^\ell a \to 1$, but $ba^r a \not\to b$. However, the next lemma, a sort of converse of Lemma 2, holds for arbitrary types.

**Lemma 3.** *Let $t_1 \ldots t_m \to T'$, $s_1 \ldots s_n \to 1$. Then $t_1 \ldots t_i s_1 \ldots s_n t_{i+1} \ldots t_m \to T'$.*

The next two lemmas state sufficient conditions for a simple type to remain present in every irreducible form of the original type. We say that $t_i t_j$ is a *block* of $T = t_1 \ldots t_m$ if $i < j$ and $t_i$ is immediately followed by $t_j$ in every irreducible form of $T$.

**Lemma 4.** *Let $t_1 \ldots t_m = T$ be modest and $i < j$ positions satisfying*

    1)   $t_i t_j \not\to 1$ and $t_{i+1} \ldots t_{j-1} \to 1$
    2)   for all $k < i$, $t_k$ is not a left adjoint of $t_i$
    3)   for all $l > j$, $t_l$ is not a right adjoint of $t_j$.

*Then $t_i t_j$ is a block of $T$. In particular, $T$ does not reduce to a single simple type nor the empty type.*

**Lemma 5.** *Suppose $T \to T'$ and $t$ occurs at $k$ distinct positions in $T$ but neither the right nor the left adjoint of $t$ occurs in $T$. Then $t$ has $k$ occurrences in $T'$.*

**Lemma 6.** *Assume that $T = t_1 \ldots t_n$ is modest and that $T \to t$. Then there is a unique position $i$ such that $t = t_i$.*
*Moreover, $t_{i+1} \ldots t_n \to 1$. In addition, $t_{i+1} \ldots t_{i+k} \to 1$ implies $t_{i+k+1} \ldots t_n \to 1$, for all $1 < k \le n-i$. Similarly, $t_1 \ldots t_{i-1} \to 1$. In addition, $t_{i-k} \ldots t_{i-1} \to 1$ implies $t_1 \ldots t_{i-k-1} \to 1$, for all $1 < k \le i - 1$.*

## 2.2 Semantic Pregroup Grammars

In a semantic pregroup grammar, see (Preller 2007-1) [18], each lexical entry $w : T$ is enriched by a (string of) logical expression(s) $E$, yielding a triple $w : T :: E$, in analogy with the triples $word : Type :: Term$ of CCG's by (Steedman 1996) [25]. The interpretation of a sentence is a variable-free expression, computed from the chosen interpretation of the words. The result of the computation depends on the chosen reduction to the sentence type. Semantic pregroup grammars are compositional in the sense of (Kracht 2007) [7]. This is best explained by replacing the (string of) logical expression(s) associated to an entry

by the corresponding 2-cell of compact 2-categories, a proof that is beyond the scope of this paper. Consider instead the following example sentences

$$\begin{array}{ll} \textit{Eva promised Jan to come} & \text{(Subject Control)} \\ \textit{Eva asked Jan to come} & \text{(Object Control)} \ . \end{array} \tag{4}$$

The implicit agent of the infinitive is either the subject of *promised* or the object of *asked*. In the first sentence it is Eva who is supposed to come, in the second it is Jan. Consider the following semantical dictionary

$$\begin{array}{llll} \textit{Eva} & : NP & :: \texttt{eva} \\ \textit{promised} & : NP^r \boldsymbol{s} \delta \bar{\boldsymbol{i}}^\ell NP^\ell & :: \texttt{promise}(x_1, x_2, x_3) & \texttt{id}(x_1) \\ \textit{asked} & : NP^r \boldsymbol{s} \delta \bar{\boldsymbol{i}}^\ell NP^\ell & :: \texttt{ask}(x_1, x_2, x_3) & \texttt{id}(x_3) \\ \textit{Jan} & : NP & :: \texttt{jan} \\ \textit{to} & : \bar{\boldsymbol{i}} \boldsymbol{i}^\ell & :: \texttt{to}(y) \\ \textit{come} & : \boldsymbol{i} \delta^r & :: \texttt{come}(z) \end{array}$$

The basic types $NP$, $\boldsymbol{i}$ and $\bar{\boldsymbol{i}}$ stand for noun phrase, infinitive and infinitival phrase. Finally, $\delta$ is a basic type that plays the role of a marker similar to an index in HPS Grammars of (Pollard, Sag) [16].

Models interpret all logical expressions as functions, including 0-*ary* functions like $\texttt{eva}$ and $\texttt{jan}$. Some functions take their values in a 'set of truth values $\Omega$' like $\texttt{promise}$, $\texttt{ask}$, $\texttt{come}$. Classical models interpret $\Omega$ as the two-element Boolean algebra and distributed models as a subset of real numbers.

Functional symbols correspond to basic types in the order in which they occur. Variables correspond to occurrences of non-basic types, indexed in the order of the occurrences of the types. For example,

$$\begin{array}{ccccc} NP^r & \boldsymbol{s} & \delta & \bar{\boldsymbol{i}}^\ell & NP^\ell \\ x_1 & \texttt{ask} & \texttt{id} & x_2 & x_3 \end{array} \ .$$

The variables on which a logical expression depends render the intuitive meaning of semantical dependency. The translations of *promised* : $NP^r \boldsymbol{s} \delta \bar{\boldsymbol{i}}^\ell NP^\ell$ and *asked* : $NP^r \boldsymbol{s} \delta \bar{\boldsymbol{i}}^\ell NP^\ell$ differ by the variable on which the translation $\texttt{id}$ of the basic type $\delta$ depends, namely on $x_1$ in the case of *promise* and on $x_3$ in the case of *ask*.

The links of a reduction to the sentence type indicate how the variables are to be replaced. For computing the logical expression corresponding to

$$\begin{array}{cccccc} \textit{Eva} & \textit{promised} & \textit{Jan} & \textit{to} & \textit{come} \\ (NP) & (NP^r \ \boldsymbol{s} \ \delta \ \bar{\boldsymbol{i}}^\ell \ NP^\ell) & (NP) & (\bar{\boldsymbol{i}} \ \boldsymbol{i}^\ell) & (\boldsymbol{i} \ \delta^r) \end{array}$$

do the following
- write the corresponding logical symbols above the simple types[1]

$$\begin{array}{cccccc} \texttt{eva} & x_1 \ \texttt{promise} \ \texttt{id} \ x_2 \ x_3 & \texttt{jan} & \texttt{to} \ y & \texttt{come} \ z \\ NP \ (NP^r & \boldsymbol{s} & \delta \ \bar{\boldsymbol{i}}^\ell \ NP^\ell) \ NP & (\bar{\boldsymbol{i}} \ \boldsymbol{i}^\ell) & (\boldsymbol{i} \ \delta^r) \end{array}$$

---

[1] Recall: a basic type $b$ is identified with the simple type $b^{(0)}$

- omit the types and put the links under the corresponding logical symbols

$$\underline{\text{eva } x_1} \; \texttt{promise} \; \underline{\text{id } x_2} \; \underline{x_3 \; \texttt{jan}} \; \underline{\texttt{to } y} \; \underline{\texttt{come}} \; z$$

- define the substitutions according to the links

$$x_1 \mapsto \texttt{eva} \quad x_2 \mapsto \texttt{to}(y) \quad x_3 \mapsto \texttt{jan} \quad y \mapsto \texttt{come}(z) \quad z \mapsto \texttt{id}(x_1) \qquad (5)$$

Substituting in $\texttt{promise}(x_1, x_2, x_3)$, one obtains the logical expression that translates the sentence

$$\texttt{promise}(\texttt{eva}, \texttt{to}(\texttt{come}(\texttt{id}(\texttt{eva}))), \texttt{jan})$$

The meaning of $\texttt{id}$ is determined by the logic, i.e. it is interpreted in every model as the identity function. This is guaranteed by the axiom

$$\texttt{id}(x) = x$$

Finally, the translation is equivalent to the variable-free expression

$$\texttt{promise}(\texttt{eva}, \texttt{to}(\texttt{come}(\texttt{eva})), \texttt{jan}) \qquad (6)$$

Note that $\texttt{eva}$ is the agent of $\texttt{come}$.

The procedure applied to the second sentence

$$\begin{array}{ccccc} \textit{Eva} & \textit{asked} & \textit{Jan} & \textit{to} & \textit{come} \\ (\underline{NP}) & (\underline{NP^r}\, \boldsymbol{s}\, \delta\, \underline{\bar{\boldsymbol{i}}^\ell\, NP^\ell}) & (\underline{NP}) & (\underline{\bar{\boldsymbol{i}}\, \boldsymbol{i}^\ell}) & (\underline{\boldsymbol{i}\, \delta^r}) \end{array},$$

yields the same substitutions as in (5) except for the last which is replaced by

$$z \mapsto \texttt{id}(x_3) \,.$$

The resulting interpretation of the sentence is now equivalent to

$$\texttt{ask}(\texttt{eva}, \texttt{to}(\texttt{come}(\texttt{jan})), \texttt{jan}) \qquad (7)$$

Now, $\texttt{jan}$ is the agent of $\texttt{come}$ in opposition to (6).

The semantical dependency, expressed above as embedding of subexpressions corresponds to the embedding of boxes in the DR-structures in (Kamp, Reyle 1993) [**?**].

## 3   Polynomial Pregroup Grammars

Polynomial pregroup grammars generalize the notion of finite pregroup grammars in (Buszkowski 2001) [1]

**Definition 5.** *A pregroup grammar is* polynomial *of degree $n$ if the length of $T_1 \ldots T_l$ is $O(l^n)$ for every every type assignment $w_1 : T_1, \ldots, w_l : T_l$ for which $T_1 \ldots T_l \to \boldsymbol{s}$.*

If the length of types occurring in the dictionary does not exceed a constant $\alpha$, then the corresponding grammar is *linear* polynomial, i.e. of degree 1. Indeed, for every string of words $w_1 \ldots w_l$ the length of the assigned type $T_1 \ldots T_l$ is bounded by the $\alpha l$. A fortiori, finite pregroup grammars are polynomial of degree 1. Hence all context free languages are generated by polynomial pregroup grammars of degree 1. The grammars for the semilinear mildly context sensitive formal languages below are also linear polynomial. The context sensitive natural language fragments considered in subsection 3.2 are generated by a square polynomial. In fact, the latter is the polynomial used in [9] for proving non semilinearity of languages.

## 3.1 Mildly Context Sensitive Formal Languages

Consider the three standard mildly context sensitive formal languages, namely

$$\mathcal{L}_1 = \{vv : v \in \Sigma^+\}$$
$$\mathcal{L}_2 = \{a^n b^n c^n : n \geq 1, a, b, c \in \Sigma, \, a \neq b, b \neq c\}$$
$$\mathcal{L}_3 = \{a^m b^n c^m d^n : m \geq 1, n \geq 1, a, b, c, d \in \Sigma, \, a \neq b, \, b \neq c, \, c \neq d\}$$

### 3.1.1 Duplication

$$\mathcal{L}_1 = \{vv : v \in \Sigma^+\}$$

The set of basic types is constructed from $\Sigma$ by adding a new symbol $\boldsymbol{s}$ called *sentence type* and a 'copy' $\bar{a}$ for every $a \in \Sigma$. The elements of $\{\boldsymbol{s}\} \cup \{\bar{a} : a \in \Sigma\}$ are pairwise distinct symbols not in $\Sigma$. The set of basic types $\mathcal{B}_1 = \langle B_1, = \rangle$ is ordered by equality, where

$$B_1 = \{\boldsymbol{s}\} \cup \Sigma \cup \{\bar{a} : a \in \Sigma\}.$$

The dictionary $\mathcal{D}_1$ maps an element $a \in \Sigma$ to the following infinite subset of $\mathcal{P}(\mathcal{B}_1)$

$$\mathcal{D}_1(a) = \{a\} \cup \{\bar{a}^r\} \cup \left\{b_1^r \ldots b_i^r a^r \boldsymbol{s} \, \bar{b}_1 \ldots \bar{b}_i : \; b_1 \ldots b_i \in \Sigma^*\right\}.$$

This dictionary is modest (recall Definition 4).

**Lemma 7.** *The language $\mathcal{L}_{\mathcal{G}_1}$ of the pregroup grammar $\mathcal{G}_1 = \langle \mathcal{D}_1, \boldsymbol{s} \rangle$ contains $\mathcal{L}_1$.*

*Proof.* Assume that $X = a_1 \ldots a_m \in \mathcal{L}_1$. Hence $m = 2n$ and $a_{n+i} = a_i$, $1 \leq i \leq n$, for some $n \geq 1$.

Case $n = 1$:
Choose $T_1 = a_1$ and $T_2 = a_1^r \boldsymbol{s}$. From the assumption follows that $T_1, T_2$ is a type assignment for $a_1 a_2$. Clearly, $T_1 T_2 \to \boldsymbol{s}$.

Case $n \geq 2$:
Define

$$T_j = \begin{cases} a_j & \text{for } 1 \leq j \leq n \\ a_n^r \ldots a_2^r a_j^r \boldsymbol{s} \, \bar{a}_n \ldots \bar{a}_2 & \text{for } j = n+1 \\ \bar{a}_j^r & \text{for } n+1 < j \leq 2n. \end{cases}$$

The assumption implies $\bar{a}_j = \bar{a}_{n+j}$ and thus $\bar{a}_j \bar{a}_{n+j}^r \to 1$, for $1 \leq j \leq n$. Hence $T_1 \ldots T_m \to s$. Thus the language $\mathcal{L}_1$ is included in the language defined by $\mathcal{G}_1$. $\qquad\square$

The type assignment $T_1, \ldots, T_m$ defined above is called the *canonical type assignment* and the concatenated type $T_1 \ldots T_m$ the *canonical type*. The unique index $k$ such that the sentence type occurs in $T_k$ is called the *key-index*.

**Lemma 8.** *Suppose $T_j \in \mathcal{D}_1(a_j)$, $1 \leq j \leq m$, is a type assignment for $X = a_1 \ldots a_m \in \Sigma^*$ such that $T_1 \ldots T_m \to s$. Then $X \in \mathcal{L}_1$ and $T_1 \ldots T_m$ is the canonical type assignment for $X$.*

*Proof.* Two things are to be proved: t $m = 2n$ for some $n \geq 1$ and that $a_{n+i} = a_i$ for $1 \leq i \leq n$. The assumption $T = T_1 \ldots T_m \to s$ implies that $s$ has a unique occurrence in $T$, by Lemma 6. Therefore there is a unique $k$ such that $1 \leq k \leq m$ and $s$ occurs in $T_k$. Hence

$$T_k = b_1^r \ldots b_i^r a_k{}^r \, s \, \bar{b}_1 \ldots \bar{b}_i$$

for some $i \in \mathbb{N}$ and some string $b_1 \ldots b_i \in \Sigma^*$. Moreover, if $j \neq k$ then $T_j = a_j$ or $T_j = \bar{a}_j^r$. From Lemma 6 follows $\bar{b}_1 \ldots \bar{b}_i T_{k+1} \ldots T_m \to 1$ and $T_1 \ldots T_{k-1} b_1^r \ldots b_i^r a_k{}^r \to 1$.

Under the assumption that $\bar{b}_1 \ldots \bar{b}_i T_{k+1} \ldots T_m \to 1$, use induction on $i$ to show that

$$\begin{aligned}
&T_{k+j} = \bar{a}_{k+j}^r \text{ for } 1 \leq j \leq i \\
&a_{k+j} = b_{i-j+1} \text{ for } 1 \leq j \leq i \\
&i = m - k \geq 0 \, .
\end{aligned} \tag{8}$$

Case $i = 0$:
Then $T_k = a_k^r s$ and $T_{k+1} \ldots T_m \to 1$, by Lemma 6. Clearly, the empty string is the only string of simple types in $\{a, \bar{a}^r : a \in \Sigma\}$ that reduces to 1. Thus $k = m$.
Case $i \geq 1$:
Note that $T_{k+1} = \bar{a}_{k+1}^r$, because the other possible choice for $T_{k+1}$ would be $a_{k+1}$. In this case $\bar{b}_i a_{k+1}$ would be in every irreducible form of $T_1 \ldots T_m$ by Lemma 4, contradicting the assumption. For the same reason, $\bar{b}_i \bar{a}_{k+1}^r \to 1$, i.e. $\bar{b}_i = \bar{a}_{k+1}$. The latter implies $a_{k+1} = b_i$. Let $T' = \bar{b}_1 \ldots \bar{b}_{i-1} T_{k+2} \ldots T_m$. Then $T' \to 1$ by Lemma 2. The induction hypothesis applies to $T'$. Hence $T_{k+1+j} = \bar{a}_{k+1+j}^r$, $a_{k+1+j} = b_{i-1-j+1}$ for $1 \leq j \leq i-1$ and $i-1 = m-(k+1)$.

Similarly, under the assumption $T_1 \ldots T_{k-1} b_1^r \ldots b_i^r a_k{}^r \to 1$ show that

$$\begin{aligned}
&T_{k-l} = a_{k-l} \text{ for } 1 \leq l \leq i+1 \\
&a_{k-l} = b_l \text{ for } 1 \leq l \leq i \\
&a_{k-i-1} = a_k \\
&1 = k - i - 1
\end{aligned} \tag{9}$$

by induction on $i$. In the case $i = 0$, note that $k > 1$, because if $k = 1$ then $a_k^r \to 1$, which is impossible. It follows that $T_{k-1} = a_{k-1}$ and $a_{k-1} = a_k$. Hence $T_1 \ldots T_{k-2} \to 1$ and therefore $k - 1 = 1$. The induction step is similar to that given above.

From equations (9) and (8) follows that $i = k-2$ and $m = k+i = k+k-2 = 2n$, where $n = k - 1$. Moreover, if $j$ varies between 1 and $i$ in increasing order then $l = i - j + 1$ varies between $i$ and 1 in decreasing order. Hence $a_{j+1} = a_{2+j-1} = a_{k-i+j-1} = b_{i-j+1} = a_{k+j} = a_{n+j+1}$, for $1 \leq j \leq i = n - 1$. Finally, $a_1 = a_{n+1}$ follows from $a_{k-i-1} = a_k$. $\qquad\square$

The proof above shows that an arbitrary type assignment with a reduction to the sentence type is equal to the canonical one and also constructs the unique reduction to the sentence type, namely

$$a_1 \; a_2 \; \ldots \; a_{k-1} \qquad\qquad a_k \qquad a_{k+1} \; \ldots \; a_m$$
$$\underline{a_1 \; \underline{a_2 \; \ldots \; \underline{a_{k-1} \; (b_1^r \ldots b_i^r)} \; a_k^r}} \; s \; \underline{\bar{b}_1 \ldots \bar{b}_i) \; \bar{a}_{k+1}^r \; \ldots \; \bar{a}_m^r} \, .$$

### 3.1.2   Multiple Agreement

$$\mathcal{L}_2 = \{ a^n b^n c^n : n \geq 1, a, b, c \in \Sigma, \; a \neq b, b \neq c \}$$

In a formal language, it is customary to denote $a^n \in \Sigma^+$ the string consisting of $n$ repetitions of the symbol $a$. This might lead to confusion because of the notation $a^{(m)}$ for simple types. Therefore the $n$-fold repetition of $t$ is denoted $[n]t$ below.

The set of basic types $\mathcal{B}_2$ is ordered by equality, where

$$B_2 = \{s\} \cup \Sigma \cup \{\bar{a} : a \in \Sigma\}$$

The dictionary $\mathcal{D}_2$ maps an element $c \in \Sigma$ to the following infinite subset of $\mathcal{P}(\mathcal{B}_2)$

$$\mathcal{D}_2(c) = \{c\} \cup \{\bar{c}^r\} \cup \left\{ [n]a^r \, [n]b^r \, s \, \bar{c}^\ell [n]\bar{c} : 1 \leq n, \; a, b \in \Sigma, \; a \neq b, \; b \neq c \right\} .$$

Note that $\mathcal{D}_2$ is modest. Moreover, $\mathcal{G}_2 = \langle \mathcal{D}_2, s \rangle$ generates $\mathcal{L}_2$.

**Lemma 9.** *A string $X = a_1 \ldots a_m \in \Sigma^*$ has a type assignment $a_i : T_i \in \mathcal{D}_2$, $1 \leq i \leq m$, such that $T_1 \ldots T_m \rightarrow s$ if and only if $X \in \mathcal{L}_2$.*

*Proof.* - Assume $X = a_1 \ldots a_m \in \mathcal{L}_2$. Then $m = 3n$ for some integer $n \geq 1$. Define

$$T_i = \begin{cases} a_i & \text{for } 1 \leq i \leq 2n \\ [n]a_{n+1}^r \, [n]a_1^r \, s \, \bar{a}_i^\ell [n]\bar{a}_i & \text{for } i = 2n + 1 \\ \bar{a}_i^r & \text{for } 2n + 2 \leq i \leq m \end{cases} .$$

Clearly, this type assignment has a reduction to the sentence type. Call it the *canonical* type assignment and $m - n + 1$ the *key-index*.

- Assume that $T_i \in \mathcal{D}_2(a_i)$, $1 \leq i \leq m$, satisfies $T_1 \ldots T_m \rightarrow s$. The argument is similar to that of Lemma 8. Now the unique type with an occurrence of the sentence type has the form

$$T_k = [n]b^r \, [n]a^r \, s \, \bar{a}_k^\ell [n]\bar{a}_k,$$

for some $k \leq m$, $n \geq 1$, $a \in \Sigma$, $b \in \Sigma$. Recalling that $\bar{a}_k^\ell [n] \bar{a}_k = \bar{a}_k^\ell \bar{a}_k [n-1] \bar{a}_k \to [n-1] \bar{a}_k$ show that

  $T_i = \bar{a}_i^r$ and $a_i = a_k$, for $k+1 \leq i \leq m$, and $n-1 = m-k$
  $T_i = a_i = a$, for $k-n \leq i \leq k-1$,
  $T_i = a_i = b$, for $1 \leq i \leq k-n-1$ and $k = 2n+1$.

From this conclude that $m = k+n-1 = 3n$. Hence $k = 2n+1$ and $X = a^n b^n a_{2n+1}^n$. $\qquad\square$

### 3.1.3 Crossing Dependencies

$$\mathcal{L}_3 = \{a^m b^n c^m d^n : m \geq 1, n \geq 1, a,b,c,d \in \Sigma,\ a \neq b,\ b \neq c,\ c \neq d\}$$

The set of basic types remains unchanged, i.e.

$$B_3 = \{\boldsymbol{s}\} \cup \Sigma \cup \{\bar{a} : a \in \Sigma\}\ .$$

The dictionary $\mathcal{D}_3$ maps an element $d \in \Sigma$ to the following infinite subset of $\mathcal{P}(\mathcal{B}_3)$

$$\mathcal{D}_3(d) = \quad \{d\} \cup \left\{\bar{d}^r\right\} \cup$$
$$\left\{ [m]c^r [n]\bar{b}^r [m]a^r \boldsymbol{s}\, \bar{d}^\ell [n]\bar{d} : n,m \geq 1\ a,b,c \in \Sigma,\ a \neq b,\ b \neq c,\ c \neq d \right\}$$

Again, $\mathcal{D}_3$ is modest and $\mathcal{G}_3 = \langle \mathcal{D}_3, \boldsymbol{s} \rangle$ generates $\mathcal{L}_3$.

**Lemma 10.** $X = a_1 \ldots a_l \in \Sigma^*$ has a type assignment $T_i \in \mathcal{D}_3(a_i)$, $1 \leq i \leq l$, such that $T_1 \ldots T_l \to \boldsymbol{s}$ if and only if $X \in \mathcal{L}_3$.

*Proof.* Assume $X = a_1 \ldots a_l = a^m b^n c^m d^n$. This implies $a_1 = \cdots = a_m = a$, $a_{m+1} = \cdots = a_{m+n} = b$, $a_{m+n+1} = \cdots = a_{2m+n} = c$, $a_{2m+n+1} = \cdots = a_l = d$ and $l = 2m + 2n$. The key-index is $k = 2m+n+1$ and the canonical type assignment satisfies

$$T_i = \begin{cases} a_i & \text{for } 1 \leq i \leq 2m+n \\ \bar{a}_i^r & \text{for } 2m+n+2 \leq i \leq l \end{cases}$$
$$T_k = [m]a_{m+n+1}^r [n]a_{m+1}^r [m]a_1^r\, s\, \bar{a}_k^\ell [n]\bar{a}_k\ .$$

The converse is left to the reader. $\qquad\square$

Note the common features shared by the three grammars $\mathcal{G}_1$-$\mathcal{G}_3$. Sentences $w_1 \ldots w_l$ have a canonical type assignment $T_1, \ldots, T_l$ with a key-index $k$. Moreover, the length of the type $T_k$ is proportional to $l$ whereas the length of the other types $T_i$, $i \neq k$, is bounded by a constant. Therefore, the length $q$ of the canonical type $t_1 \ldots t_q = T_1 \ldots T_l$ is $O(l)$.

### 3.2 Natural Languages

Among the context sensitive natural language fragments are the Dutch and Swiss-German subordinate clauses and the compound noun phrases of Old Georgian.

*Dutch Subordinate Clause*

(Pullum, Gazdar) [20] presents a context free grammar that weakly generates the Dutch subordinate clauses. According to Salvitch *et al.*) [**?**]sa, the context free grammar generates the clauses as strings of symbols but produces parse trees that violate the intuition of speakers about the phrase structure and the semantical dependencies. On the other hand, (Bresnan *et al.*) [2] argues that no context free grammar strongly generates the clauses.

The polynomial pregroup grammar below strongly generates the Dutch subordinate clauses. This means that the reductions to the sentence type give rise to a semantic interpretation expressing the distant cross-dependencies. For example, in the subordinate clause[2] below *Marie* is the agent of *zag* and *Jan* the agent of *zwemmen*. The dependency is represented by an arrow from the verb to the agent.

$$\overset{\frown}{dat\ Marie\ Jan\ zag\ zwemmen} \tag{10}$$

*(that Mary saw Jan swim)*

Using the entries

$$
\begin{array}{llll}
dat & (\text{that}) & : \boldsymbol{s}\bar{\boldsymbol{s}}^{\ell} & :: \texttt{dat}(y) \\
Marie & (\text{mary}) & : NP & :: \texttt{marie} \\
zag & (\text{saw}) & : NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^{\ell}\delta & :: \texttt{zien}(x_2, z) \quad \texttt{id}(x_1) \\
Jan & (\text{jan}) & : NP & :: \texttt{jan} \\
zwemmen & (\text{swim}) & : \delta^r \boldsymbol{i} & :: \texttt{zwemmen}(x) \quad,
\end{array}
$$

parse this clause

$$
\begin{array}{cccc}
dat\ Marie\ Jan & zag & zwemmen \\
\texttt{dat marie jan} & (\ x_1 \quad x_2\ \texttt{zien}\ z\ \texttt{id}) & (\ x\ \texttt{zwemmen}) \\
\boldsymbol{s}\,\bar{\boldsymbol{s}}^{\ell}\ (NP)\,(NP)\,(NP^r\,NP^r & \bar{\boldsymbol{s}} & \bar{\boldsymbol{i}}^{\ell}\ \delta)\ (\delta^r\,\bar{\boldsymbol{i}})
\end{array} \tag{11}
$$

and compute its logical interpretation according to Section 2.2

$$\texttt{dat}(\texttt{zien}(\texttt{marie}, \texttt{zwemmen}(\texttt{id}(\texttt{jan})))).$$

Applying the identity axiom $\texttt{id}(\texttt{jan}) = \texttt{jan}$, we see that the interpretation of the clause is equivalent to

$$\texttt{dat}(\texttt{zien}(\texttt{marie}, \texttt{zwemmen}(\texttt{jan}))). \tag{12}$$

By convention, the first argument of a relation corresponds to the agent. Hence, the subexpression relation in (12) expresses the semantical dependencies of sentence (10).

---

[2]adapted from examples in (Bresnan *et al.*) [2]

The dependency arrows of (10) can also be obtained geometrically. It suffices to represent the dependencies by curved overlinks. The vertical arrows indicate the functional symbols, connected to their arguments by the dotted overlinks.

$$\text{marie jan } x_1\, x_2 \text{ zien } z \text{ id } x \text{ zwem men .} \tag{13}$$
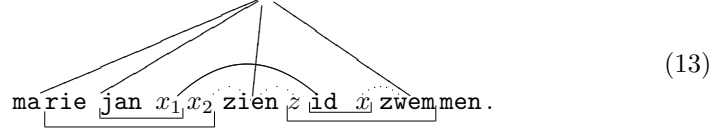
In (13), the path starting at $x$ and ending at $\mathtt{jan}$ and the path from $x_2$ to $\mathtt{marie}$ constitute the dependency links of (10). The dependence of $\mathtt{zien}$ on $x_2$ and $z$ is indicated by dotted overlinks.[3]
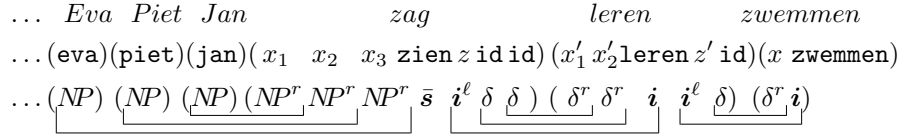
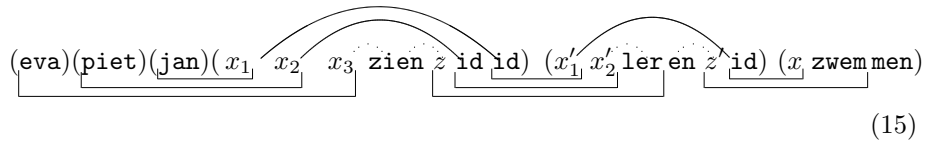The number of noun phrases and causal verbs is not limited, for example

$$\textit{(dat) Eva Piet Jan zag leren zwemmen} \tag{14}$$

$$\textit{((that) Eva saw Pete teach Jan to swim)}$$
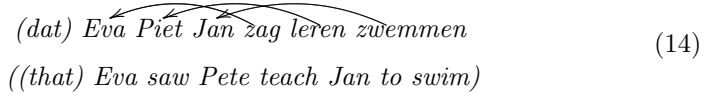
Consider the entries $Eva : NP :: \mathtt{eva}$, $Piet : NP :: \mathtt{piet}$, $Jan : NP :: \mathtt{jan}$ and

| | | | |
|---|---|---|---|
| $zag$ | $\left(_{\text{saw}}\right)$ | $: NP^r NP^r NP^r \bar{s} i^\ell \delta \delta$ | $:: \mathtt{zien}(x_3, z) \quad \mathtt{id}(x_2)\ \mathtt{id}(x_1)$ |
| $leren$ | $\left(_{\text{teach}}\right)$ | $: \delta^r \delta^r i i^\ell \delta$ | $:: \mathtt{leren}(x_2', z')\ \mathtt{id}(x_1')$ |
| $zwemmen$ | $\left(_{\text{swim}}\right)$ | $: \delta^r i$ | $:: \mathtt{zwemmen}(x)$ |

Compute the reduction of the assigned type to $\bar{s}$

$$\dots\ Eva\ Piet\ Jan \qquad zag \qquad\qquad leren \qquad zwemmen$$

$$\dots (\mathtt{eva})(\mathtt{piet})(\mathtt{jan})(x_1\quad x_2\quad x_3\ \mathtt{zien}\ z\ \mathtt{id}\ \mathtt{id})(x_1'\ x_2' \mathtt{leren}\ z'\ \mathtt{id})(x\ \mathtt{zwemmen})$$

$$\dots (NP)\ (NP)\ (NP)(NP^r\, NP^r\, NP^r\ \bar{s}\quad i^\ell\ \delta\ \delta\ )\ (\ \delta^r\, \delta^r\quad i\quad i^\ell\ \delta)\ (\delta^r\, i)$$

Replace the simple types by the corresponding logical symbols and represent dependencies by curved overlinks

$$(\mathtt{eva})(\mathtt{piet})(\mathtt{jan})(x_1\quad x_2\quad x_3\ \mathtt{zien}\ z\ \mathtt{id}\ \mathtt{id})\ (x_1'\ x_2'\ \mathtt{ler\,en}\ z'\ \mathtt{id})\ (x\ \mathtt{zwem\,men}) \tag{15}$$

The oriented paths starting at $x_3$, $x_2'$ and $x$ and terminating at $\mathtt{eva}$, $\mathtt{piet}$ and $\mathtt{jan}$ respectively constitute the dependency arrows of (14). The logical expression is

$$\mathtt{zien}(\mathtt{eva}, \mathtt{leren}(\mathtt{piet}, \mathtt{zwemmen}(\mathtt{jan}))) .$$

---

[3]This vindicates Claudia Casadio's idea that overlinks intervene in grammatical dependencies. The graph above the logical symbols in (13) represents the concatenation of the meanings of the words. It 'lives' in symmetric compact *2*-categories, like the category of real vector spaces. The overlinks correspond to expansions in the symmetric 2-category, but not in the non-symmetric 2-category of derivations of the Pregroup Calculus. The meaning of the sentence is obtained by composing the concatenated meanings with the reduction.

The graph (15) induces the labelled planar graph (16) belonging to a family of graphs relevant for dependency parsing, see (Kuhlmann, Nivre) [8]. The labels of the edges in (16) are defined by the overlinks 'hidden' inside of (the type of) the words. Every path formed by the edges with a given label corresponds to a dependency arrow of (14).

$$\textit{Eva} \quad \textit{Piet} \quad \textit{Jan} \quad \textit{z a g} \quad \textit{leren zwemmen} \tag{16}$$
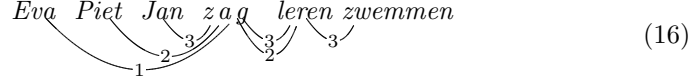
The last verb in the clause may be intransitive, transitive, ditransitive *etc.* The *arity* of a verb $w$ is the number of the argument places of the interpreting relation. Hence intransitive verbs are of arity 1, transitive verbs are of arity 2 and so on. Note that the arity of a non-causal verb coincides with the number of occurrences of non-basic types, for example

$$
\begin{array}{llll}
\textit{zwemmen} & (\text{swim}) : \delta^r \boldsymbol{i} & :: \texttt{zwemmen}(x) & (\text{intransitive}) \\
\textit{schrijven} & (\text{write}) : \delta^r \delta^r \boldsymbol{i} & :: \texttt{schrijven}(x_2, x_1) & (\text{transitive}) \\
\textit{geven} & (\text{give}) \ : \delta^r \delta^r \delta^r \boldsymbol{i} :: \texttt{geven}(x_3, x_2, x_1) & (\text{ditransitive})
\end{array}
$$

The arity of the causal verbs below also is 2. The surplus number of non-basic types in an associated type $T_p$, $p \geq 2$, provides the argument places for the 'remembering' functions id. For example,

$$
\begin{array}{lll}
\textit{zag} & (\text{saw}) : [p]NP^r \bar{\boldsymbol{s}} \boldsymbol{i}^\ell [p-1]\delta :: \texttt{zien}(x_p, z) & \texttt{id}(x_{p-1}) \dots \texttt{id}(x_1), \ p \geq 2 \\
\textit{leren} & (\text{teach}): [p]\delta^r \boldsymbol{i}\boldsymbol{i}^\ell [p-1]\delta \quad :: \texttt{leren}(x_p, z) & \texttt{id}(x_{p-1}) \dots \texttt{id}(x_1), \ p \geq 2
\end{array}
\tag{17}
$$

where $x_1$ corresponds to the first occurrence of $NP^r$, $x_2$ to the second occurrence of $NP^r$ and so on up to $x_p$, whereas $z$ corresponds to $\boldsymbol{i}^\ell$.

A string of words $w_1 \dots w_l$ is a *k-fold subordinate clause* if its first word $w_1$ is *dat*, the words $w_2$ up to and including $w_{1+k}$ are proper names, the next word $w_{1+k+1}$ is a causal verb in finite form, the so-called *key-word*, and after the key-word the $w_i$'s are infinitives, of which all are causal except the last one, which is non-causal of arity $m = 2k + 2 - l$.

A $k$-fold subordinate clause $w_1 \dots w_l$ has a canonical type assignment $T_i$, $1 \leq i \leq l$, namely

$$
T_i = \begin{cases}
\boldsymbol{s}\,\bar{\boldsymbol{s}}^\ell & \text{if } i = 1 \\
NP & \text{if } 2 \leq i \leq k+1 \\
[k]NP^r \bar{\boldsymbol{s}}\,\boldsymbol{i}^\ell[k-1]\delta & \text{if } i = k+2 \\
[2k-i+2]\delta^r \boldsymbol{i}\,\boldsymbol{i}^\ell[2k-i+1]\delta & \text{if } k+2 < i < l \\
[2k-l+2]\delta^r \boldsymbol{i} & \text{if } i = l \qquad .
\end{cases}
$$

The proof that the canonical type assignment reduces to the clause type $\boldsymbol{s}$ uses induction on $k$ and follows from the next two lemmas.

A type $T$ is said to be *p-infinitival* if either $T = [p]\delta^r \boldsymbol{i}$ and $p \geq 1$ or $T = [p]\delta^r \boldsymbol{i}\boldsymbol{i}^\ell [p-1]\delta$ and $p \geq 2$. It is said to be *causal* if the latter holds and non-causal in the former case.

14

**Lemma 11.** *Let $T_j$ be infinitival or equal to NP for $1 \leq j \leq n$. Then $T_1 \ldots T_n \nrightarrow 1$*

*Proof.* Assume on the contrary that $T_1 \ldots T_n \to 1$. As $\delta^\ell$ and $NP^\ell$ do not occur in $T_1 \ldots T_n$ the latter does not end with $\delta$ nor with $NP$. Hence $T_n = [p_n]\delta^r \boldsymbol{i}$. Therefore the number of occurrences of $\boldsymbol{i}$ in $T_1 \ldots T_n$ exceeds that of $\boldsymbol{i}^\ell$, because the latter always occurs together with the former. As $\boldsymbol{i}$ can only be linked to $\boldsymbol{i}^\ell$, this contradicts $T_1 \ldots T_n \to 1$. $\square$

**Lemma 12.** *Let $k \geq 1$, $n \geq 1$, $Z = \boldsymbol{i}^\ell[k]\delta$ and $T_j$ be $p_j$-infinitival of length $q_j$, $1 \leq j \leq n$ such that $ZT_1 \ldots T_n \to 1$ holds. Then*

   i)    *$T_n$ is non-causal*
   ii)   *$T_j$ is causal, $1 \leq j \leq n-1$. Moreover, $p_j = k - j + 1$, $1 \leq j \leq n$*
   iii)  *$n = k - p_n + 1$*
   iv)  *$q_j = 3 + 2(k-j)$, $1 \leq j < n$, $q_n = k - n + 2$.*

*Proof.* From $ZT_1 \ldots T_n \to 1$ follows that $T_n = [p_n]\delta^r \boldsymbol{i}$ by the same argument as above. Hence i) holds.
Next show ii), iii) and iv) by induction on $n$.
    Case $n = 1$. From $\boldsymbol{i}^\ell[k]\delta[p_1]\delta^r \boldsymbol{i} \to 1$ follows that $k = p_1$
    Case $n \geq 2$. Recall that $ZT_1 \ldots T_n = \boldsymbol{i}^\ell[k]\delta[p_1]\delta^r \boldsymbol{i}XY \to 1$ where $Y = T_2 \ldots T_n$ and either $X = 1$ or $X = \boldsymbol{i}^\ell[p_1 - 1]\delta$. The latter alternative holds if $T_1$ is causal, the former if it is non-causal. Note that $\boldsymbol{i}^r$ does not occur in the string $XY$ and therefore the leftmost occurrence of $\boldsymbol{i}$ in $ZT_1 \ldots T_n \to 1$ is linked to the unique $\boldsymbol{i}^\ell$ on its left. It follows that $[k]\delta[p_1]\delta^r \to 1$ and $XY \to 1$, hence $k = p_1$. If $X = 1$ then $T_2 \ldots T_n = Y \to 1$, contradicting Lemma 11 and the assumption $n \geq 2$. Hence $X = \boldsymbol{i}^\ell[p_1 - 1]\delta$ with $p_1 \geq 2$. Now apply the induction hypothesis to $X, T_2, \ldots, T_n$.
Finally, iii) and iv) are immediate consequences of ii) and i). $\square$

**Theorem 1.** 1) *For every subordinate clause there is a unique type assignment with a reduction to the clause type $\boldsymbol{s}$. 2) Every string of words from the dictionary that has a type assignment with a reduction to $\boldsymbol{s}$ is a subordinate clause.*

*Proof.* 1) The first assertion follows from the definitions by Lemmas 1 - 6.
    2) To see the converse, let $T_j \in \mathcal{D}(w_j)$ for $1 \leq j \leq l$ and $T_1 \ldots T_l \to \boldsymbol{s}$. By Lemma 5, $\boldsymbol{s}$ occurs in exactly one type $T_i$ and therefore $T_i = \boldsymbol{s}\bar{\boldsymbol{s}}^\ell$ and $w_i = dat$. Then $\bar{\boldsymbol{s}}$ also has exactly one occurrence in the string. Indeed, each of its occurrences is linked to some occurrence of $\bar{\boldsymbol{s}}^\ell$ and the latter occurs only together with $\boldsymbol{s}$. Let $p$ be the unique index such that $\bar{\boldsymbol{s}}$ occurs in $T_p$. Therefore $T_j$ is either infinitival or the basic type $NP$ for all $j$ other than $i$ and $p$. Note that $p > i$, because $\bar{\boldsymbol{s}}^\ell$ and $\bar{\boldsymbol{s}}$ are linked. Moreover, $T_p = [k]NP^r \bar{\boldsymbol{s}} \boldsymbol{i}^\ell[k-1]\delta$ for some $k \geq 2$.
    First note that $i = 1$, because $T_1 \ldots T_{i-1} \to 1$ by choice of $i$. This is only possible if the string is empty by Lemma 11.
    Next, $T_2 \ldots T_{p-1}[k]NP^r \to 1$, because $\bar{\boldsymbol{s}}^\ell$ is linked to $\bar{\boldsymbol{s}}$. From this follows that $T_2 = \cdots = T_{p-1} = NP$ and $p = 1 + k + 1$ by Lemma 4.

Finally, from the preceding follows that $\boldsymbol{i}^{\ell}[k-1]\delta\, T_{p+1}\ldots T_l \to 1$. Note that *NP* cannot occur in this string and conclude by Lemma 12. ☐

Theorem 1 above implies that for every $\boldsymbol{s}$-sentence $w_1\ldots w_l$ there are unique types $T_i \in \mathcal{D}(w_i)$, $1 \le i \le l$, and a unique type $T = T_1\ldots T_l$ such that $T \to \boldsymbol{s}$. Call $T$ the *canonical type*, $T_i \in \mathcal{D}(w_i)$ the *canonical type assignment* and the unique reduction of $T$ to $\boldsymbol{s}$ the *canonical reduction* of $w_1\ldots w_l$.

The preceding theorem implies that the infinite grammar above is *polynomial*, i.e. the length of any type assignment with a derivation to $\boldsymbol{s}$ is bounded by a polynomial. The property also intervenes in the complexity estimate of the parsing algorithm in Section 4.

**Corollary 2.** *The length of the canonical type $T_1\ldots T_l$ of a $k$-fold Dutch subordinate clause is bounded by $k^2 + 3k + 1$. Moreover, $k \le l/2$.*

*Proof.* Let $k$ be the number of noun phrases preceding the key-word $w_p$, $m$ the arity of the last verb $w_l$. The number $n$ of words after $w_p$ satisfies $n = k - m$ by Lemma 12. The number of words before the key-word $w_p$ is $k + 1$. Hence $l = 2k - m + 2$ and therefore $k \le l/2$.

On the other hand, the length $q$ of the type $T_1\ldots T_l$ is

$$q = 2 + k + q' + m + 1,$$

where $q'$ is the length of the type $T_p T_{p+1}\ldots T_{l-1}$. Starting at $T_{l-1}$ and reading backward from right to left, the length of the types increases by 2 from one to the next. The length of the rightmost type $T_{l-1}$ is $2 + 2m + 1$. Therefore

$$
\begin{aligned}
q' \;&= \textstyle\sum_{j=1}^{j=n}(2j + 2m + 1)\\
&= n(2m + 1) + n(n + 1)\\
&= k^2 + 2k - m^2 - 2m
\end{aligned}
$$

Hence, $q = k^2 + 3k - m^2 - m + 3 \le k^2 + 3k + 1$. ☐

The canonical reduction defines the semantic dependencies as well. This follows from the next lemma, where a path in the oriented graph $\mathcal{G}$ represents the successive substitutions and instances of the identity axiom intervening in the interpretation.

**Lemma 13.** *Let $k > n \ge 0$ and $\mathcal{G} = \langle \mathcal{V}_0 \cup \mathcal{V}_1, \mathcal{E}_0 \cup \mathcal{E}_1 \rangle$ be the oriented graph defined as follows*
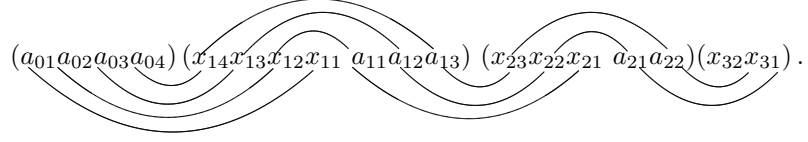
$$
\begin{aligned}
\mathcal{V}_0 =\ & \{a_{ij} : 0 \le i \le n,\, 1 \le j \le k - i\} && \text{(functional symbol)}\\
\mathcal{V}_1 =\ & \{x_{lp} : 1 \le l \le n+1,\, 1 \le p \le k - l\} && \text{(variable)}
\end{aligned}
$$

*and*

$$
\begin{aligned}
\mathcal{E}_0 =\ & \{\langle a_{ij}, x_{i+1,j} \rangle : 0 \le i \le n,\ 1 \le j \le k - i\} && \text{(substitution)}\\
\mathcal{E}_1 =\ & \{\langle x_{il}, a_{i,l-1} \rangle : 1 \le i \le n,\ 2 \le l \le k - i\} && \text{(identity axiom)} \quad .
\end{aligned}
$$

*Then for $1 \le j \le k$ there is a unique maximal path starting at $a_{0j}$. Moreover, $x_{l,j+1-l}$ and $a_{i,j-i}$ are on this path for all $l$ such that $1 \le l \le n+1$ and $j-l \ge 0$ and all $i$ satisfying $j-i \ge 1$ and $0 \le i \le n$.*

*Proof.* Straightforward by induction on $k$. The graph looks like this for $k = 4, n = 2$, where the underlinks represent the edges in $\mathcal{E}_0$ and the overlinks the edges in $\mathcal{E}_1$

$$(a_{01}a_{02}a_{03}a_{04})\,(x_{14}x_{13}x_{12}x_{11}\ a_{11}a_{12}a_{13})\ (x_{23}x_{22}x_{21}\ a_{21}a_{22})(x_{32}x_{31})\,.$$

$\square$

## Pregroup Grammar with Copying Rules

(Stabler 2004) [23] considers copying grammars when analysing crossing dependencies in human languages. Following this lead, define the following finite pregroup grammar enriched with two copying rules

$$
\begin{array}{lll}
\text{dictionary entries} & \text{copying rules} & \\
zag \quad (\text{saw}): NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell \delta & NP^r T\delta \to NP^r NP^r T\delta\delta & (18) \\
leren\ (\text{teach}): \delta^r \delta^r \boldsymbol{i}\boldsymbol{i}^\ell \delta & \delta^r T\delta \to \delta^r \delta^r T\delta\delta &
\end{array}
$$

The copying rules are not derivable in Pregroup Calculus. Therefore the graphical representations of derivations is lost and with them the mathematical structure of the proofs. Moreover, the semantical interpretation of the control verbs cannot be read off the type in the dictionary but must be constructed during the proof by a semantic copying rule parallel to the grammatical copying rule.

It is easy to see that every clause recognized by the polynomial pregroup grammar is also recognized by the finite grammar with copying rules. Indeed, let $T_1, \ldots, T_n$ be a type assignment for $w_1, \ldots, w_n$ from the infinite dictionary with a reduction $r$ to the clause type. The corresponding type assignment $T'_j, 1 \leq j \leq n$ from the finite dictionary has the following derivation to the clause type. For every type $T_j$ of the form $[p_j]NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell [p_j - 1]\delta$ respectivley $[p_j]\delta^r \boldsymbol{i}\boldsymbol{i}^\ell [p_j - 1]\delta$ apply $p_j - 2$ times the coppying rule to $T'_j = NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell \delta$ respectivley to $T'_j = \delta^r \delta^r \boldsymbol{i}\boldsymbol{i}^\ell \delta$. The resulting compound type is $T_1, \ldots, T_n$. Then apply the contraction rule as indicated by the links of $r$.

The converse also holds. A string derivable in the copying grammar also has a reduction in the polynomial grammar. First show that an arbitrary derivation can be replaced by one where all copying rules are applied before the contraction rule. Then use an argument similar to that establishing the second part of Theorem 1.

## Swiss German Subordinate Clause

According to the analysis of (Shieber) [22], the Swiss-German subordinate clause has the same semantic cross-serial dependencies as Dutch, but they are also expressed in the syntax by case marking. This can be captured by distinguishing the types for noun phrases $NP_{\text{nom}}, NP_{\text{dat}}, NP_{\text{acc}}$ as well as the dummy types

$\delta_{\mathrm{nom}}, \delta_{\mathrm{dat}}, \delta_{\mathrm{acc}}$. The proofs are similar to the preceding ones. In particular, correct semantical dependencies guarantee correct syntax.

## Old Georgian Noun Phrase

The pregroup grammar below generates compound noun phrases of Old Georgian according to the analysis of [9]. The dictionary lists an infinite number of distinct words. Indeed, Old Georgian uses genitive suffixes for possessive compound noun phrases. The genitive suffix is appended to noun(stem)s or names. When the construction is repeated, the previous genitive suffixes are also repeated.

$$\begin{array}{cccccc} \textit{govel-i} & \textit{igi} & \textit{sisxl-i} & \textit{saxl-isa-j} & \textit{m-is} & \textit{Saul-is-isa-j} \\ \texttt{all-Nom} & \texttt{Art=Nom} & \texttt{blood-Nom} & \texttt{house-Gen-Nom} & \texttt{Art-Gen} & \texttt{Saul-Gen-Gen-Nom} \end{array} \quad (19)$$

'all the blood of the house of Saul'

More generally, compound nominative noun phrases have the form

$$\texttt{N}_1\texttt{-Nom } \texttt{N}_2\texttt{-Gen-Nom } \texttt{N}_3\texttt{-Gen}^2\texttt{-Nom}\ldots\texttt{N}_k\texttt{-Gen}^{k-1}\texttt{-Nom}. \quad (20)$$

Square brackets highlight semantic dependencies as follows

$$[\texttt{N}_1\texttt{-Nom } [\texttt{N}_2\texttt{-Gen-Nom } [\texttt{N}_3\texttt{-Gen}^2\texttt{-Nom}\ldots[\texttt{N}_k\texttt{-Gen}^{k-1}\texttt{-Nom}]_{NP_k}\cdots]_{NP_3}]_{NP_2}]_{NP_1}.$$

Assume the basic types $NP_{\mathrm{nom}}$, $N_{\mathrm{nom}}$ and $G$ for nominative noun phrases, nominative common nouns and genitive suffixes in that order. For each $p \geq 0$, the word $\textit{Name}\texttt{-Gen}^p\texttt{-Nom}$ respectively $\textit{Noun}\texttt{-Gen}^p\texttt{-Nom}$ has two entries in the dictionary, namely

$$\begin{aligned} \textit{Name}\texttt{-Gen}^p\texttt{-Nom} &: \begin{cases} NP_{\mathrm{nom}}\,[p]G \\ NP_{\mathrm{nom}}\,[p]G\,[p+1]G^\ell\,NP_{\mathrm{nom}}^\ell \end{cases} \\ \textit{Noun}\texttt{-Gen}^p\texttt{-Nom} &: \begin{cases} N_{\mathrm{nom}}\,[p]G \\ N_{\mathrm{nom}}\,[p]G\,[p+1]G^\ell\,NP_{\mathrm{nom}}^\ell. \end{cases} \end{aligned} \quad (21)$$

Common nouns are preceded[4] by a determiner to form noun phrases like

$$\begin{array}{ll} \textit{Art} = \texttt{Nom} & \textit{Noun}\texttt{-Nom} \\ \textit{Art}\texttt{-Gen} & \textit{Noun}\texttt{-Gen}^p\texttt{-Nom}. \end{array} \quad (22)$$

Adding the following entries to the dictionary

$$\begin{array}{ll} \textit{Art} = \texttt{Nom} : NP_{\mathrm{nom}}\,N_{\mathrm{nom}}^\ell \\ \textit{Art}\texttt{-Gen} \quad : NP_{\mathrm{nom}}GG^\ell N_{\mathrm{nom}}^\ell, \end{array} \quad (23)$$

each of the noun phrases (22) has two possible types, each of which reduces to the type of a noun phrase. For $p \geq 1$, the types for $\textit{Art}\texttt{-Gen } \textit{Noun}\texttt{-Gen}^p\texttt{-Nom}$ are

$$(NP_{\mathrm{nom}}GG^\ell N_{\mathrm{nom}}^\ell)\,(N_{\mathrm{nom}}\,[p]G) \to NP_{\mathrm{nom}}[p]G,$$
$$(NP_{\mathrm{nom}}GG^\ell N_{\mathrm{nom}}^\ell)\,(N_{\mathrm{nom}}\,[p]G\,[p+1]G^\ell\,NP_{\mathrm{nom}}^\ell) \to NP_{\mathrm{nom}}\,[p]G\,[p+1]G^\ell\,NP_{\mathrm{nom}}^\ell.$$

---

[4]The determiner may also follow its noun. This is ignored here.

The length of the type at the left of $\rightarrow$ exceeds that of the reduced type by 4.

It follows that every string of words of the form (20) has a unique type assignment with a reduction to the noun phrase type *NP* and *vice versa*. The length of the assigned type can be expressed as a square polynomial in the length of the string.

## 4   Tagging and Parsing Algorithm

Ambiguity enters parsing by pregroup grammars in two ways. There may be different type assignments with a reduction to the sentence type or a fixed type assignment of length $q$ may have (up to $2^q$) distinct reductions to the sentence type. Testing every type assignment for an eventual reduction to the sentence type is highly inefficient even if the dictionary is finite. The usual cubic-time polynomial recognition algorithms do not construct reductions and rely on the fact that the dictionary is finite or at least that there is a constant bounding the number of types per word in the dictionary. Some authors use 'parsing' in the weak sense that the algorithm constructs a reduction to the sentence type for a given type assignment, whereas the choice of a type assignment is called 'tagging'. 'Parsing' is used here in the following stronger sense.

**Definition 6.** *A recognition algorithm decides whether or not a string of words $w_1 \ldots w_l \in \Sigma^*$ has a type assignment $T_1, \ldots, T_l$ such that the concatenated type $T_1 \ldots T_l$ has a reduction to the sentence type. A parsing algorithm is an algorithm that decides whether or not a string of words is a sentence and, if the answer is yes, computes a type assignment and a reduction to the sentence type.*

Recognition is sufficient for formal languages, but parsing is indispensable for natural languages, because the semantic interpretation of a sentence is defined via the derivation to the sentence type.

The algorithm below is a variant of the algorithm in (Preller 2007-2) [19]. It processes the string of words from left to right and chooses a type for each word. The choice relies on a tagging strategy motivated by properties specific to the languages $\mathcal{L}_i$ of the preceding section. The strategy avoids 'losing' type-assignments as soon as possible. That is to say it avoids a type assignment $T_1, \ldots, T_i$ that cannot be extended to a type assignment $T_1, \ldots, T_i, T_{i+1}, \ldots, T_l$ with a reduction to the sentence type.

### 4.1   The Algorithm

A *stack* of non-negative integers is defined inductively. The empty symbol $\varnothing$ is a stack, called the empty stack. If $S'$ is a stack and $i$ a non-negative integer then $\langle S', i \rangle$ is a stack. The functions *top* and *pop* send a stack $\langle S', i \rangle$ to its *top* $i$ and to its *tail* $S'$ respectively. They are undefined for the empty stack.

When processing the string $w_1 \ldots w_l \in \Sigma^*$, the algorithm moves through a

subset of the following set of *stages*

$$
\begin{aligned}
\mathcal{S}_{w_1 \ldots w_l} = \quad & \{(i; T_1, \ldots, T_i; p) : 1 \le i \le l,\ 1 \le p \le q_i,\ T_j \in \mathcal{D}(w_j),\ 1 \le j \le i\} \\
& \cup \{(i; T_1, \ldots, T_{i-1}, 1; 0) : 1 \le i \le l+1,\ T_j \in \mathcal{D}(w_j),\ 1 \le j \le i-1\} \\
& \cup \{s_{in}\},
\end{aligned}
$$

where $q_i$ is the length of the type $T_i$ and $s_{in}$ is a new symbol denoting the *initial* stage.

A partial order is defined on the set of stages as follows

$$
\begin{aligned}
s_{in} \le \quad & s && \text{for all } s \\
(i; T_1, \ldots, T_i; p) \le \quad & (i'; T_1', \ldots, T_{i'}'; p') && \Leftrightarrow i < i',\ T_j = T_j' \text{ for } 1 \le j \le i \\
& && \text{or} \\
& && i = i',\ p = 0,\ T_j = T_j' \text{ for } 1 \le j < i \\
& && \text{or} \\
& && i = i',\ 1 \le p \le p',\ T_j = T_j',\ 1 \le j \le i\,.
\end{aligned}
$$

This partial order induces a total order on the set of all stages less or equal to a given stage. Moreover, all stages of the form $(l+1; T_1, \ldots, T_l, 1; 0)$ are maximal.

Every non-initial stage $s$ has a unique *predecessor* $s - 1$, given by

$$
(i; T_1, \ldots, T_i; p) - 1 = \begin{cases}
(i; T_1, \ldots, T_i; p-1), & \text{if } p \ge 2 \\
(i; T_1, \ldots T_{i-1}, 1; 0) & \text{if } p = 1 \\
(i - 1; T_1, \ldots, T_{i-1}; q_{i-1}) & \text{if } p = 0 \text{ and } i > 1 \\
s_{in} & \text{if } p = 0 \text{ and } i = 1\,.
\end{cases}
$$

A non-initial stage is *tagging* if its last integer $p = 0$ and *testing* otherwise. Every testing stage $s$ has a unique *successor* $s + 1$, namely

$$
(i; T_1, \ldots, T_i; p) + 1 = \begin{cases}
(i; T_1, \ldots, T_i; p+1) & \text{if } p < q_i \\
(i + 1; T_1, \ldots, T_i, 1; 0) & \text{if } p = q_i \text{ and } i \le l\,.
\end{cases}
$$

The algorithm executes two subroutines, *tagging* and *testing*, for each word $w_i$. When at tagging stage $(i; T_1, \ldots, T_{i-1}, 1; 0)$, $i \le l$, the algorithm has finished processing the type $T_1 \ldots T_{i-1}$. The tagging routine $tag_{\mathcal{D}}$ either chooses a type $Tag \in \mathcal{D}(w_i)$ or decides to stop and updates the constant *output*. The computation of $Tag$ involves a constant *key* that depends on the language whose sentences are to be parsed. The routine $tag_{\mathcal{D}}$ is defined in the next subsection. At a maximal stage $(l + 1; T_1, \ldots, T_l; 0)$, the output is updated to the result computed so far.

Recall that the types $T_j$ at stage $s = (i; T_1, \ldots, T_i; p)$ are strings of simple types $T_j = t_{j1} \ldots t_{jq_j}$. Each testing stage $s = (i; T_1, \ldots, T_i; p)$ defines a *working position* $p(s) = q_1 + \cdots + q_i + p$, the simple type *read* $t_{p(s)} = t_{ip}$ and the type *processed* $T(s) = t_1 \ldots t_{p(s)} = t_{11} \ldots t_{1q_1} \ldots t_{i1} \ldots t_{ip}$. To keep notation uniform, define $p(s_{in}) = 0$ and $t_0 = 1$. Note that for every testing stage $s$ and positive integer $i' \le p(s)$, there is a unique testing stage $s' \le s$ such that $i' = p(s')$.

20

When in testing stage $s$, the algorithm checks if $t_{p(s)}$ contracts with the last not yet contracted simple type and updates the stack of positions $S(s)$ and the reduction $\mathcal{R}(s)$. The latter contains the links computed so far. The former contains the unlinked positions in increasing order such that the top of the stack is the position of the last unlinked simple type.

Finally, the irreducible substring $I(s)$ of $T(s)$ consisting of the unlinked simple types in the order given by the stack $S(s)$ is defined by

$$I(s_{in}) = 1,\ I(\langle S', j \rangle) = I(S')t_j\,.$$

**Definition 7. Tagger-Parser**

▼ At the initial stage $s = s_{in}$, *key, output, $S$* and $\mathcal{R}$ are initialized to

$$key = undefined,\ output = undefined,\ S(s_{in}) = \langle \varnothing, 0 \rangle,\ \mathcal{R}(s_{in}) = \emptyset\,.$$

Then the process goes to the first tagging stage

$$s = (1; 1; 0)$$

▼ At tagging stage $s = (i; T_1, \ldots, T_{i-1}, 1; 0)$, the stack and reduction remain unchanged

$$S(s) = S(s-1),\ \mathcal{R}(s) = \mathcal{R}(s-1)\,.$$

If $i = l + 1$ the process is in a maximal stage and updates *output*

$$output = \langle \mathcal{R}(s-1), T(s-1), I(s-1) \rangle$$

If $i \leq l$, the next type $T_i$ is chosen

$$\begin{array}{c} tag_{\mathcal{D}}(i) \\ T_i = Tag \end{array}$$

and the process goes to the next stage unless $tag_{\mathcal{D}}(i)$ updates *output* to *fail*

$$\text{if } output \neq fail \text{ then } s = (i; T_1 \ldots, T_{i-1}, T_i; 1)$$

▼ At testing stage $s = (i; T_1, \ldots, T_i; p),\ p \geq 1$,

$$S(s) = \begin{cases} pop(S(s-1)) & \text{if } t_{top(S(s-1))}t_{p(s)} \to 1 \\ \langle S(s-1), p(s) \rangle & \text{else} \end{cases}$$

$$\mathcal{R}(s) = \begin{cases} \mathcal{R}(s-1) \cup \{\{top(S(s-1)), p(s)\}\} & \text{if } t_{top(S(s-1))}t_{p(s)} \to 1 \\ \mathcal{R}(s-1) & \text{else} \end{cases}$$

Then the process goes to the next stage

$$s = s + 1\,.$$

The proof of the next lemma is given in (Preller 2007-2) [19].

**Lemma 14.** *For every stage $s = (i; T_1, \ldots, T_i; p)$, the string of simple types $I(s)$ associated to the stack $S(s)$ is an irreducible substring of $T(s)$ and $\mathcal{R}(s)$ is a reduction from $T(s)$ to $I(s)$.*

## 4.2 Tagging Strategy

The strategy is based on the fact that a sentence has a unique derivation to the sentence type. The strategy chooses the type a word must have if the whole string is a sentence. Testing validates or invalidates that choice.

In the case of the formal language $\mathcal{L}_1$, the key index for $w_1 \ldots w_l$ is $l/2 + 1$. If the latter is not an integer, the string is not a sentence. At the first tagging stage $(1; 1; 0)$, processing is stopped if the length $l$ of the string is odd.

$tag_{\mathcal{D}_1}(i)$
if $i = 1$
  if $l/2 \neq \lceil l/2 \rceil$ let $output=fail$
  else $key = l/2 + 1$
if $i < key$ let $Tag = w_i$
if $i = key$ let $Tag = w_{i-1}^r \ldots w_2^r w_i^r \boldsymbol{s}\, \bar{w}_{i-1} \ldots \bar{w}_2$
if $i > key$ let $Tag = \bar{w}_i^r$

In the case of $\mathcal{L}_3$, the routine $tag_{\mathcal{D}_3}$ tests whether the length is even. If this not the case the string $w_1 \ldots w_l$ is not a sentence. Otherwise, it computes the number $m$ of repetitions of the first word $w_1$. If $w_1 \ldots w_l$ is a sentence the number $n$ of repetitions of $w_{m+1}$ satisfies $n = l/2 - m$ and the key satisfies $key = 2m + n + 1$.

$tag_{\mathcal{D}_3}(i)$
if $i = 1$
  if $l/2 \neq \lceil l/2 \rceil$ let $output=fail$
  else let $Tag = w_i$
if $i > 1$ and $key = undefined$
  let $Tag = w_i$
  if $w_i \neq w_{i-1}$ let $key = l/2 + i$ , $m = i - 1$, $n = l/2 - (i - 1)$
if $i < key$ let $Tag = w_i$
if $i = key$ let $Tag = [m]w_{m+n+1}^r[n]w_{m+1}^r[m]w_1^r \boldsymbol{s}\, \bar{w}_i^\ell[n]\bar{w}_i$
if $i > key$ let $Tag = \bar{w}_i^r$

The case of $\mathcal{L}_2$ is similar with the appropriate adaptations.

Finally, in the case of the Dutch dictionary $\mathcal{D}_4$, the types are chosen according to the following properties of a subordinate clause

- the key-index $key$ must be the first $p$ for which $w_p$ is a causal verb in finite form

- the non-causal words have a unique type in the dictionary

- every word after the key-word except the last is a causal infinitive. The last is a non-causal infinitive.

$tag_{\mathcal{D}_4}(i)$
if $key = undefined$
  if $w_i$ is an infinitive let $output=fail$
  else
  if $w_i$ is not a causal verb in finite from let $Tag \in \mathcal{D}(w_i)$

|  | $Tag$ | $\mathcal{S}$ | $\mathcal{R}$ |
|---|---|---|---|
| $s_{in}$ | $undefined$ | $\langle \varnothing, 0\rangle$ | $\varnothing$ |
| $(1;1;0)$ | $T_1 = NP$ | $\langle \varnothing, 0\rangle$ | $\varnothing$ |
| $(1;T_1;1)$ | $NP$ | $\langle \langle \varnothing, 0\rangle, 1\rangle$ | $\varnothing$ |
| $(2;T_1,1;0)$ | $T_2 = NP$ | $\langle \langle \varnothing, 0\rangle, 1\rangle$ | $\varnothing$ |
| $(2;T_1,T_2;1)$ | $NP$ | $\langle \langle \langle \varnothing, 0\rangle, 1\rangle, 2\rangle$ | $\varnothing$ |
| $(3;T_1,T_2,1;0)$ | $T_3 = NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell\delta$ | $\langle \langle \langle \varnothing, 0\rangle, 1\rangle, 2\rangle$ | $\varnothing$ |
| $(3;T_1,T_2,T_3;1)$ | $NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell\delta$ | $\langle \langle \varnothing, 0\rangle, 1\rangle$ | $\{2,3\}$ |
| $(3;T_1,T_2,T_3;2)$ | $NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell\delta$ | $\langle \varnothing, 0\rangle$ | $\{\{2,3\},\{1,4\}\}$ |
| $(3;T_1,T_2,T_3;3)$ | $NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell\delta$ | $\langle \langle \varnothing, 0\rangle, 5\rangle$ | $\{\{2,3\},\{1,4\}\}$ |
| $(3;T_1,T_2,T_3;4)$ | $NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell\delta$ | $\langle \langle \langle \varnothing, 0\rangle, 5\rangle, 6\rangle$ | $\{\{2,3\},\{1,4\}\}$ |
| $(3;T_1,T_2,T_3;5)$ | $NP^r NP^r \bar{\boldsymbol{s}}\boldsymbol{i}^\ell\delta$ | $\langle \langle \langle \langle \varnothing, 0\rangle, 5\rangle, 6\rangle, 7\rangle$ | $\{\{2,3\},\{1,4\}\}$ |
| $(4;T_1,T_2,T_3,1;0)$ | $T_4 = \delta^r\boldsymbol{i}$ | $\langle \langle \langle \langle \varnothing, 0\rangle, 5\rangle, 6\rangle, 7\rangle$ | $\{\{2,3\},\{1,4\}\}$ |
| $(4;T_1,\ldots,T_4;1)$ | $\delta^r\boldsymbol{i}$ | $\langle \langle \langle \varnothing, 0\rangle, 5\rangle, 6\rangle$ | $\{\{2,3\},\{1,4\}\}\cup \{\{7,8\}\}$ |
| $(4;T_1,\ldots,T_4;2)$ | $\delta^r\boldsymbol{i}$ | $\langle \langle \varnothing, 0\rangle, 5\rangle$ | $\{\{2,3\},\{1,4\}\}\cup \{\{7,8\},\{6,9\}\}$ |

Table 1: Processing *Marie Jan zag zwemmen*.

if $w_i$ is a causal verb in finite form and $i > 1$ let $key = i$, $k = i - 1$ and $Tag = [k]NP^r \ \bar{\boldsymbol{s}} \ \boldsymbol{i}^\ell[k-1]\delta$ else let *output=fail*
if $i > key$
  if $w_i$ is an infinitive
    of a causal verb let $p = k - (i - key)$ and $Tag = [p]\delta^r \ \boldsymbol{i} \ \boldsymbol{i}^\ell[p-1]\delta$
    of a non-causal verb let $Tag \in \mathcal{D}(w_i)$
  else *output=fail*

When fed the string of words *Marie Jan zag zwemmen* the parser goes through the following stages and values. The constant *key* remains *undefined* until a causal verb in finite form $w_i$ is encountered. There the value is updated to $i$ and remains unchanged afterwards. The values of the other constants are listed in Table 1 below. At the maximal stage $s_{fin} = (5;T_1,\ldots,T_4,1;0)$, the *output* is updated to

$$\begin{aligned} \mathcal{R}(s_{fin}) &= \{\{2,3\},\{1,4\},\{7,8\},\{6,9\}\} \\ T(s_{fin}) &= NP\,NP\,NP^r\,NP^r\,\bar{\boldsymbol{s}}\boldsymbol{i}^\ell\delta\delta^r\boldsymbol{i} \\ I(s_{fin}) &= \bar{\boldsymbol{s}} \end{aligned}$$

One could argue following (Lambek 2008-2) [13] that simple types represent bits of information to be stored in the short-term memory (limited to $7\pm2$ bits) when processing a string of words. The present algorithm seems to confirm this claim. Theoretically, a Dutch speaker can form subordinate clauses of arbitrary

length. In practice, three to four noun phrases between *dat* and the causal verb in finite form are rarely exceeded. For two noun phrases, the stack contains at most four bits. With three noun phrases, it goes up to five, with four to six. Accepting that the types represent the patterns in which a word can appear, one also accepts that they are learned in childhood and 'hard-wired'. This includes the pattern of causal verbs represented by $[p]NP^r \boldsymbol{si}^\ell[p-1]\delta$, where $p$ is 2 or more. They are downloaded to the 'working' memory where the subconscious processing goes on and only the result ends up in the short-term memory.

**Theorem 3.** *The string of words $w_1 \ldots w_l$ is a sentence if and only if the tagging-parsing algorithm reaches a maximal stage such that output $= \langle \mathcal{R}, T, \boldsymbol{s} \rangle$. If this is the case, $\mathcal{R}$ is a reduction of $T$ to $\boldsymbol{s}$. Moreover, the algorithm is linear for the formal languages and square polynomial for the natural languages.*

*Proof.* The first assertion follows from Lemma 14 and Lemma 1. Moreover, the number of basic steps executed at a testing stage is bounded by a constant. It is proportional to the length of the chosen type at a non-maximal tagging stage. Finally, updating the output at a maximal stage $s_{fin}$ is proportional to the length of $T(s_{fin})$. □

## 5   Conclusion

It may be worth-while to investigate whether the degree of the polynomial of a pregroup grammar can serve as a classification for (natural) languages. Indeed, proof-search in the Pregroup Calculus is bounded by a cubic polynomial in the length of types. Therefore in general, the search for a derivation is cubic polynomial in the length of the type even *after* type assignment. This is in opposition to categorial grammars based on Syntactic Calculus where proof-search is NP-complete, (Pentus 2003) [15]. Hence, the ratio of the length of the concatenated type over the number of words is essential when designing a pregroup grammar with an efficient algorithm.

The parsing complexity for the languages considered here is lower than the general cubic polynomial limit because proof-search is linear for the sets of types occurring in the dictionaries and because the algorithm constructs a single derivation while processing from left to right. Proof-search remains linear for larger classes of types than those mentioned here. This gives rise to grammars for language fragments involving relative pronouns and coordination, subject and object control, agreement of features among others. In fact, these grammars have been designed for a complete linear deterministic parsing algorithm with occasional backtracks producing a planar dependency graph. Empirical studies based on large scale treebanks in (Nivre) [14] show that such algorithms are highly accurate for other formalisms in general where no proof of completeness exists.

# References

[1] Bhatt Rajesh, Joshi Aravind (2004). Semilinearity is a syntactic invariant: A reply to Michaelis and Kracht 1997. Linguistic Inquiry: 35, pp. 683-692

[2] Bresnan Joan, Ronald M. Kaplan, Stanley Peters and Annie Zaenen (1987), Cross-Serial Dependencies in Dutch, *in*: The Formal Complexity of Natural Language, Walter Salvitch, *ed.*, Reidel Publishing Company, pp. 286-319

[3] Buszkowski Wojciech ( 2001), Lambek Grammars based on pregroups, in: P. de Groote *et al.*, editors, Logical Aspects of Computational Linguistics, LNAI 2099, Springer LIRMM N 03O24, 2003

[4] Buszkowski Wojciech (2002), Cut elimination for the Lambek calculus of adjoints, in: Abrusci et al., (*eds*), *Papers in formal linguistics and logic*, Bulzoni, Bologna

[5] Francez Nissim, Michael Kaminski (2007), Commutation Augmented Pregroup Grammars and Mildly Context Sensitive Languages, *in*: W. Buszkowski and A. Preller, editors, Studia Logica, Vol.87, issue 2:3, pp. 295-321, Springer
doi:10.1007/s11225-007-9088-z

[6] Kamp Hans, Uwe Reyle (1993), *From Discourse to Logic, Introduction to Modeltheoretic Semantics of Natural Language*, Kluwer Academic Publishers

[7] Kracht Marcus (2007), Compositionality:"The Very Idea, Research in Language and Computation", 5:287-308

[8] Kuhlmann Marco, Joakim Nivre(2006), Midly Non-Projective Dependency Structure, Proceedings of the COLING/ACL on Main conference poster sessions, pp. 507- 514

[9] Michaelis Jens, Marcus Kracht (1997), Semilinearity as a syntactic invariant,*in*: Ch. Retoré, editors, Logical Aspects of Computational Linguistics, Springer Lecture Notes in Computer Science, 1328, pp. 329-345

[10] Lambek Joachim (1958), The mathematics of sentence structure, American Mathematical Monthly 65, pp.154-170, 1958

[11] Lambek Joachim (1999), Type Grammar revisited, *in*: Alain Lecomte et al., editors, Logical Aspects of Computational Linguistics, Springer LNAI 1582, pp.1-27

[12] Lambek Joachim (2008), From word to sentence, Polimetrica, Milano, Italy 2008, ISBN 978-88-7699-117-2

[13] Lambek Joachim (2008), Reflections on English Pronouns, in: Casadio, Claudia and Joachim Lambek (eds.), *Computational and Algebraic Approaches to Natural Language*, pp. 233-253 , Polimetrica, Milano, Italy

[14] Nivre Joakim (2008), Algorithms for deterministic incremental dependency parsing, Computational Linguistics, Volume 34, Issue 4, 513-553

[15] Pentus Mati (2003), Lambek calculus is NP-complete, CUNY ph.D. Program in Computer Science Technical Report TR-2003005, CUNY Graduate Center, New York
http://www.cs.gc.cuny.edu/tr/techreport.php?id=79

[16] Pollard Carl and Ivan A. Sag, Head-driven phrase structure grammar, The University of Chicago Press, 1994

[17] Preller Anne, Joachim Lambek (2007), Free compact 2-183-226, 1987 categories, Mathematical Structures for Computer Sciences, 17(1), 1-32

[18] Preller Anne (2007), Toward Discourse Representation Via Pregroup Grammars, JoLLI, Vol.16, pp. 173-194
doi:10.1007/s10849-006-9033-y

[19] Preller Anne (2007), Linear Processing with Pregroup Grammars, *in*: W. Buszkowski *et al.*, editors, Studia Logica, vol. 87, issue 2:3, pp. 171 - 197
doi:10.1007/s11225-007-9087-0

[20] Pullum Geoffrey, Gerald Gazdar (1987), Natural Languages and Context Free Languages, in: The Formal Complexity of Natural Language, Walter Salvitch, ed., Reidel Publishing Company, pp. 138-182

[21] Salvitch Walter, Emmon Bach, William Marsh and Gila Safran-Naveh (1987), Introduction to Part III, *in*: The Formal Complexity of Natural Language, Walter Salvitch *et al.*, ed., Reidel Publishing Company, pp. 283-285

[22] Shieber Stuart M. (1987), Evidence Against the Context-Freeness of Natural Language, *in*: The Formal Complexity of Natural Language, Walter Salvitch, *ed.*, Reidel Publishing Company, pp. 320-334

[23] Stabler Edward (2004), Varieties of crossing dependencies: Structure dependence and mild context sensitivity, Cognitive Science 28(5), pp. 669-720

[24] Stabler Edward (2008), Tupled Pregroup Grammars, in Claudia Casadio, Joachim Lambek (eds.),*Computational and Algebraic Approaches to Natural Language*, Polimetrica, Milano, Italy

[25] Steedman Mark (1996), Surface Structure and Interpretation, Cambridge Mass.: MIT Press, Linguistic Inquiry Monograph, 30