Polynomial Pregroup Grammars parse Context Sensitive Languages
Anne Preller

To cite this version:

HAL Id: lirmm-00538562
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00538562
Submitted on 22 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Polynomial Pregroup Grammars parse Context Sensitive Languages

ANNE PRELLER
LIRMM/CNRS, Montpellier, France

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 by Lambek (1999) as a simplification of the earlier Syntactic Calculus in (Lambek, 1958). According to Buszkowski (2001), 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 and Kaminski (2008) and Stabler (2008) extend pregroup grammars to mildly context-sensitive formal languages by adding new rules and/or constraints to the Pregroup Calculus. Lambek (2008a) discusses the Dutch subordinate clause and remarks that a law of commutativity would solve the problem, but dismisses it as ‘not allowed’.

I am thankful for financial support provided by TALN/LIRMM.
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 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 $w_i : T_i, 1 \leq i \leq l$ 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 and Kracht (1997)’s version of Old Georgian noun phrases is polynomial of degree 2. The description of the same noun phrases in (Bhatt and Aravind, 2004) 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). 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 $P(B)$ generated by a partially ordered set $B = \langle B, \leq \rangle$ is the free monoid generated by the set of simple types

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

The notation $a^{(z)}$ designates the ordered pair formed by the element $a$ of $B$ and the integer $z$. Elements $T \in P(B)$ 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

left adjoint: $t' = (a^{(z)})' = (a^{(z-1)})$
right adjoint: $t' = (a^{(z)})' = (a^{(z+1)})$.

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

(1) \begin{align*}
(\text{Induced step}) & \quad S a^{(z)} T \rightarrow S b^{(z)} T \\
(\text{Generalized contraction}) & \quad S a^{(z)} b^{(z+1)} T \rightarrow S T \\
(\text{Generalized expansion}) & \quad S T \rightarrow S a^{(z+1)} b^{(z)} T
\end{align*}

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, $a \rightarrow a$ by generalized expansion and $(aa')a \rightarrow a$ by generalized contraction, see Buszkowski (2002).

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

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

$L_G = \{ w_1 \ldots w_n : T_1 \ldots T_n \rightarrow s \text{ for some } T_i \in D(w_i), 1 \leq i \leq n \}$.

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 Pre-group Calculus can be characterized geometrically by proof graphs.
that have underlinks, overlinks and vertical links, see (Preller and Lambek, 2007). 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. The proof graph below represents a derivation from the type $a\ell a a a a a a s$ to the type $c a b s c c$.

To check grammaticality of a string of words, only derivations without instances of Generalized Expansion are to be considered, see Lambek (1999). 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 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.

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 $R$ from $T$ to $T'$, in symbols $R : T \Rightarrow T'$, is a non-directed graph $R = \langle \{1, \ldots, n\}, R \rangle$ such that

1) for $i \in \{1, \ldots, n\} - \{i_1, \ldots, i_k\}$ there is exactly one $j \leq n$ such that $\{i, j\} \in R$
2) if $\{i, j\} \in R$ then $i \neq j$ and $i \neq \{i_1, \ldots, i_k\}$, $j \notin \{i_1, \ldots, i_k\}$
3) for $\{i, j\} \in R$ and $i < l < j$ then there is $m$ such that $i < m < j$ and $\{i, m\} \in R$
4) $t_it_j \rightarrow 1$ 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 $1 \leq i \leq n$. It occurs in $\mathcal{D}$, if it occurs in the type of some lexical entry of $\mathcal{D}$.

For example, $a^l$ and $s$ occur in $sa^r$, 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 $t_it_{i+1} \not\rightarrow 1$ for $1 \leq i < n$. A type $T'$ is an irreducible form of $T$ if $T'$ is irreducible and $T \rightarrow T'$.

For example, if $a$ and $b$ are basic then $aa^rb^a$ is modest and irreducible. Every simple type is irreducible. The type constituting the top line of (2) is not modest.

The proofs of lemmas 2.2. - 2.6. below are straight forward. Lemma 2.1. is a special case of Lemma 4.5 in Preller (2007a).

Lemma 2.1. (Uniqueness) Every modest type has a unique irreducible form and a unique reduction to its irreducible form.

The property does not hold in general. Indeed, $ba'ad'a'$ has the two irreducible forms $b$ and $ba'$, whereas (2) gives a type with two distinct reductions to the same irreducible form.
**Lemma 2.2.** Let $T'$ be the irreducible form of a modest type $T = t_1 \ldots t_m$ and suppose that $t_i \ldots t_j \rightarrow 1$ for some $1 \leq i < j \leq m$. Then $t_1 \ldots t_{i-1}t_{j+1} \ldots t_m \rightarrow T'$.

Again, the property above does not hold in general. For example, $ba'a' \rightarrow b$, $a' \rightarrow 1$, but $ba' \not\rightarrow b$. However, the next lemma, a sort of converse of Lemma 2.2., holds for arbitrary types.

**Lemma 2.3.** If $t_1 \ldots t_m \rightarrow T'$ and $S = s_1 \ldots s_n \rightarrow 1$ then

$$t_1 \ldots t_iS t_{i+1} \ldots t_m \rightarrow T'.$$

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 $t_it_j$ 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 2.4.** Let $t_1 \ldots t_m = T$ be modest and $i < j$ positions satisfying

1) $t_i t_j \not\rightarrow 1$ and $t_{i+1} \ldots t_{j-1} \rightarrow 1$
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 2.5.** Suppose $T \rightarrow T'$ 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 2.6.** Assume that $T = t_1 \ldots t_n$ is modest and that $T \rightarrow t$. Then there is a unique position $i$ such that $t = t_i$.

Moreover, $t_i \rightarrow 1$. In addition, $t_{i+1} \ldots t_{i+k} \rightarrow 1$ implies $t_{i+k+1} \ldots t_n \rightarrow 1$, for all $1 < k \leq n - i$. Similarly, $t_1 \ldots t_{i-1} \rightarrow 1$. In addition, $t_{i-k} \ldots t_{i-1} \rightarrow 1$ implies $t_1 \ldots t_{i-k-1} \rightarrow 1$, for all $1 < k \leq i - 1$.

### 2.2. Semantic Pregroup Grammars

In a semantic pregroup grammar, see Preller (2007b), 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). 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). 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

Eva promised Jan to come (Subject Control)
Eva asked Jan to come (Object Control).

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

<table>
<thead>
<tr>
<th>Eva</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>promised</td>
<td>NP' sδi' Np' :: promise(x1, x2, x3) id(x1)</td>
</tr>
<tr>
<td>asked</td>
<td>NP' sδi' Np' :: ask(x1, x2, x3) id(x3)</td>
</tr>
<tr>
<td>Jan</td>
<td>NP</td>
</tr>
<tr>
<td>to</td>
<td>i'</td>
</tr>
<tr>
<td>come</td>
<td>iδr'</td>
</tr>
</tbody>
</table>

The basic types NP, i and i' stand for noun phrase, infinitive and infinitival phrase. Finally, δ is a basic type that plays the role of a marker similar to an index in HPS Grammars of Pollard and Sag (1994).

Models interpret all logical expressions as functions, including 0-ary functions like eva and jan. Some functions take their values in a ‘set of truth values Ω′ like promise, ask, come. Classical models interpret Ω 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,

\[ NP' \ s \ \delta \ i' \ NP' \]

\[ x_1 \ \text{ask} \ \text{id} \ x_2 \ x_3 \]

The variables on which a logical expression depends render the intuitive meaning of semantical dependency. The translations of
promised : \(\text{NP} s \delta \iddots \text{NP} s\) and asked : \(\text{NP} s \delta \iddots \text{NP} s\) differ by the variable on which the translation \(\text{id}\) 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

\[
\text{Eva} \quad \text{promised} \quad \text{Jan} \quad \text{to} \quad \text{come}
\]

\[
(\text{NP}) (\text{NP} s \delta \iddots \text{NP} s) (\text{NP})(\iddots \text{NP} s)(\iddots \text{id})
\]

do the following

- write the corresponding logical symbols above the simple types\(^2\)

\[
\text{eva} \quad x_1 \quad \text{promise} \quad \text{id} \quad x_2 \quad x_3 \quad \text{jan} \quad \text{to} \quad y \quad \text{come} \quad z
\]

\[
\text{NP} (\text{NP} s \delta \iddots \text{NP} s)(\text{NP})(\iddots \text{id})
\]

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

\[
\text{eva} \quad x_1 \quad \text{promise} \quad \text{id} \quad x_2 \quad x_3 \quad \text{jan} \quad \text{to} \quad y \quad \text{come} \quad z
\]

- define the substitutions according to the links

(3)

\[
\begin{align*}
x_1 & \mapsto \text{eva} \\
x_2 & \mapsto \text{to}(y) \\
x_3 & \mapsto \text{jan} \\
y & \mapsto \text{come}(z) \\
z & \mapsto \text{id}(x_1)
\end{align*}
\]

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

\[
\text{promise(\text{eva}, \text{to(\text{come(\text{id(\text{eva}))})}), \text{jan})}
\]

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

\[
\text{id}(x) = x
\]

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

(4)

\[
\text{promise(\text{eva}, \text{to(\text{come(\text{eva})}), \text{jan})}
\]

\(^2\)Recall: a basic type \(b\) is identified with the simple type \(b^{(0)}\)
Note that *eva* is the agent of *come*.

The procedure applied to the second sentence

\[
\text{Eva asked Jan to come}
\]

\[
(NP)(NP')s\delta\bar{e}i(\bar{e})'(i\delta')
\]

yields the same substitutions as in (3) except for the last which is replaced by

\[
z \mapsto \text{id}(x_3).
\]

The resulting interpretation of the sentence is now equivalent to

\[
(5) \quad \text{ask(eva, to(come(jan)), jan)}
\]

Now, *jan* is the agent of *come* in opposition to (4).

The semantical dependency, expressed above as embedding of subexpressions corresponds to the embedding of boxes in the DR-structures in citek-r.

3. Polynomial Pregroup Grammars

Polynomial pregroup grammars generalize the notion of finite pregroup grammars in (Buszkowski, 2001)

**Definition 5.** A pregroup grammar is polynomial of degree *n* if the length of *T*₁...*T*l is \(O(n^l)\) for every type assignment \(w_1 : T_1, \ldots, w_l : T_l\), for which \(T_1 \ldots T_l \rightarrow s\).

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 (Michaelis and Kracht, 1997) for proving non semilinearity of languages.
3.1. Mildly Context Sensitive Formal Languages

Consider the three standard mildly context sensitive formal languages, namely

\[ L_1 = \{ vv : v \in \Sigma^+ \} \]

\[ L_2 = \{ a^n b^n c^n : n \geq 1, a, b, c \in \Sigma, a \neq b, b \neq c \} \]

\[ L_3 = \{ a^m b^n c^m d^n : m \geq 1, n \geq 1, a, b, c, d \in \Sigma, a \neq b, b \neq c, c \neq d \} . \]

3.1.1. Duplication

\[ L_1 = \{ vv : v \in \Sigma^+ \} \]

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

\[ B_1 = \{ s \} \cup \Sigma \cup \{ \bar{a} : a \in \Sigma \} . \]

The dictionary \( D_1 \) maps an element \( a \in \Sigma \) to the following infinite subset of \( P(B_1) \)

\[ D_1(a) = \{ a \} \cup \{ \bar{a} \} \cup \{ b_1 \ldots b_i a' s \bar{a}_1 \ldots \bar{a}_i : b_1 \ldots b_i \in \Sigma^* \} . \]

This dictionary is modest (recall Definition 4.).

**Lemma 3.1.** The language \( L_{G_1} \) of the pregroup grammar \( G_1 = \langle D_1, s \rangle \) contains \( L_1 \).

**Proof.** Assume that \( X = a_1 \ldots a_m \in L_1 \). 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 \( T_2 = a'_2 s \). From the assumption follows that \( T_1, T_2 \) is a type assignment for \( a_1 a_2 \). Clearly, \( T_1 T_2 \rightarrow s \).

Case \( n \geq 2 \):

Define

\[ T_j = \begin{cases} 
- a_j & \text{for } 1 \leq j \leq n \\
- a'_j \ldots a'_i a'_j s \bar{a}_n \ldots \bar{a}_2 & \text{for } j = n + 1 \\
- \bar{a}'_j & \text{for } n + 1 < j \leq 2n .
\end{cases} \]
The assumption implies \( \bar{a}_j = \bar{a}_{n+j} \) and thus \( \bar{a}_j \bar{a}_{n+j} \to 1 \), 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 \( \mathcal{G}_1 \).

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 3.2.** Suppose \( T_j \in D_1(a_j), 1 \leq j \leq m \), 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: \( m = 2n \) for some \( n \geq 1 \) and that \( a_{n+j} = a_i \) 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 2.6. Therefore there is a unique \( k \) such that \( 1 \leq k \leq m \) and \( s \) occurs in \( T_k \). Hence

\[
T_k = b_1' \ldots b_i' a_k' s \bar{b}_1 \ldots \bar{b}_i
\]

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 \( T_j = \bar{a}_j \). From Lemma 2.6. follows that

\[
\bar{b}_1 \ldots \bar{b}_i T_{k+1} \ldots T_m \to 1
\]

\[
T_1 \ldots T_{k-1} b_1' \ldots b_i' a_k' \to 1
\]

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

\[
T_{k+j} = \bar{a}_j \quad \text{for} \quad 1 \leq j \leq i
\]

\[
a_{k+j} = b_{i-j+1} \quad \text{for} \quad 1 \leq j \leq i
\]

\[
i = m - k \geq 0
\]

Case \( i = 0 \): Then \( T_k = a_k' s \) and \( T_{k+1} \ldots T_m \to 1 \), by Lemma 2.6. Clearly, the empty string is the only string of simple types in \( \{ a, \bar{a} : a \in \Sigma \} \) that reduces to 1. Thus \( k = m \).

Case \( i \geq 1 \):

Note that \( T_{k+1} = \bar{a}_{k+1} \), 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 2.4., contradicting the assumption. For the same reason, \( \bar{b}_i \bar{a}_{k+1} \to 1 \), 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 2.2. The
induction hypothesis applies to $T'$. Hence $T_{k+1+j} = a'_{k+1+j}$, $a_{k+1+j} = b_{i-1-j+1}$ for $1 \leq j \leq i-1$ and $i-1 = m - (k+1)$.

Similarly, under the assumption $T_1 \ldots T_{k-1} b'_1 \ldots b'_k \overset{1}{\Rightarrow}$ show that

$$
T_{k-l} = a_{k-l} \text{ for } 1 \leq l \leq i + 1 \\
a_{k-l} = b_l \text{ for } 1 \leq l \leq i \\
a_{k-1} = a_k \\
1 = k - i - 1
$$

by induction on $i$. In the case $i = 0$, note that $k > 1$, because if $k = 1$ then $a'_1 \overset{1}{\Rightarrow} 1$, which is impossible. It follows that $T_{k-1} = a_{k-1}$ and $a_{k-1} = a_k$. Hence $T_1 \ldots T_{k-2} \overset{1}{\Rightarrow}$ and therefore $k - 1 = 1$. The induction step is similar to that given above.

From equations (7) and (6) 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 $a_{j+1} = a_{2+j-1} = a_{k+1+j-1} = b_{l-j+1} = a_{k+j} = a_{n+j+1}$, for $1 \leq j \leq i - 1$. Finally, $a_1 = a_{n+1}$ follows from $a_{k-1} = a_k$.

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

$$
\begin{array}{cccccccccc}
& a_1 & a_2 & \ldots & a_{k-1} & a_k & a_{k+1} & \ldots & a_m \\
\hline
a_1 & a_2 & \ldots & a_{k-1} & (b'_1 \ldots b'_k a'_{\overline{k}} s \bar{b}_1 \ldots \bar{b}_j) & & & & a'_{\overline{m}} \\
\end{array}
$$

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 $a^{(n)}$ for simple types. Therefore the $n$-fold repetition of $t$ is denoted $[n]t$ below.

The set of basic types $B_2$ is ordered by equality, where

$$
B_2 = \{s\} \cup \Sigma \cup \{\bar{a} : a \in \Sigma\}
$$
The dictionary $D_2$ maps an element $c \in \Sigma$ to the following infinite subset of $P(B_2)$

$$D_2(c) = \{c\} \cup \{[c] \mid \exists \ell \in \mathbb{N} \text{ such that } \ell \geq 2 \wedge \exists n, a, b \in \Sigma, c \notin a \neq b \neq c\}.$$ 

Note that $D_2$ is modest. Moreover, $G_2 = \langle D_2, s \rangle$ generates $L_2$.

**Lemma 3.3.** A string $X = a_1 \ldots a_m \in \Sigma^*$ has a type assignment $a_i : T_i \in D_2$, $1 \leq i \leq m$, such that $T_1 \ldots T_m \rightarrow s$ if and only if $X \in L_2$.

**Proof.** Assume $X = a_1 \ldots a_m \in L_2$. Then $m = 3n$ for some integer $n \geq 1$. Define

$$T_i = \begin{cases} 
  a_i & \text{for } 1 \leq i \leq 2n \\
  [n]a'_{i+1} \mid [n]a'_i & \text{for } i = 2n + 1 \\
  \bar{a}'_i & \text{for } 2n + 2 \leq i \leq m 
\end{cases}$$

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 $T_i \in D_2(a_i)$, $1 \leq i \leq m$, satisfies $T_1 \ldots T_m \rightarrow s$. The argument is similar to that of Lemma 3.2. Now the unique type with an occurrence of the sentence type has the form

$$T_k = [n]b' [n]a' s \bar{a}'_k [n]a_k,$$

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

$$T_i = \bar{a}'_i \text{ and } a_i = a_k, \text{ for } k + 1 \leq i \leq m, \text{ and } n - 1 = m - k$$

$$T_i = a_i = a, \text{ for } k - n \leq i \leq k - 1,$$

$$T_i = a_i = b, \text{ for } 1 \leq i \leq k - n - 1 \text{ and } k = 2n + 1.$$

From this conclude that $m = k + n - 1 = 3n$. Hence $k = 2n + 1$ and $X = a^d b^c d^n$.

**3.1.3. Crossing Dependencies**

$$L_3 = \{a^b b^c e^m d^n : m \geq 1, n \geq 1, a, b, c, d \in \Sigma, a \neq b, b \neq c, c \neq d\}$$

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

$$B_3 = \{s\} \cup \Sigma \cup \{\bar{a} : a \in \Sigma\}.$$
The dictionary $D_3$ maps an element $d \in \Sigma$ to the following infinite subset of $P(B_3)$:

$$D_3(d) = \{d\} \cup \{\bar{d}\} \cup \{[m]c^r[n]b^r[m]a^r s \bar{d}[n]\bar{d} : n, m \geq 1, a, b, c \in \Sigma\},$$

where $a \neq b$, $b \neq c$, $c \neq d$. Again, $D_3$ is modest and $G_3 = \langle D_3, s \rangle$ generates $L_3$.

**Lemma 3.4.** $X = a_1 \ldots a_l \in \Sigma^*$ has a type assignment $T_i \in D_3(a_i)$, $1 \leq i \leq l$, such that $T_1 \ldots T_l \rightarrow s$ if and only if $X \in L_3$.

**Proof.** The key-index is $k = 2m + n + 1$ and the canonical type assignment is

$$T_i = \begin{cases} a_i & \text{for } 1 \leq i \leq 2m + n \\ \bar{a}_i' & \text{for } 2m + n + 2 \leq i \leq l \end{cases}$$

$$T_k = [m]a_{m+n+1}^r [n]a_{m+1}^r [m]a_r^s \bar{d}[n]\bar{d}.$$ 

The details are left to the reader. □

Note the common features shared by the three grammars $G_1 \ldots G_3$. 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 and Gazdar (1987) presents a context free grammar that weakly generates the Dutch subordinate clauses. This means that 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, see (Salvitch et al., 1987). On the other hand, Bresnan et al. (1987) 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 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.

\[
\text{dat Marie Jan zag zwemmen}
\]

\[(that \text{ Mary saw Jan swim)}\]

Using the entries

\[
\begin{align*}
\text{dat} & \quad (\text{na}) : s\bar{s}^f \quad :: \text{dat}(y) \\
\text{Marie} & \quad (\text{naar}) : NP \quad :: \text{marie} \\
\text{zag} & \quad (\text{mo}) : NP' NP' s\bar{s}^f \bar{o} :: \text{zien}(x_2, z) \quad \text{id}(x_1) \\
\text{Jan} & \quad (\text{naar}) : NP \quad :: \text{jan} \\
\text{zwemmen} & \quad (\text{mo}) : \bar{o}\bar{r}^i \quad :: \text{zwemmen}(x) \\
\end{align*}
\]

parse this clause

\[
\text{dat Marie Jan zag zwemmen}
\]

\[
\text{dat marie jan }, (x_1 x_2 \text{ zien id })(x \text{ zwemmen})
\]

and compute its logical interpretation according to Section 2.2

\[
\text{dat (zien (marie, zwemmen (id (jan)))).}
\]

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

\[
\text{dat (zien (marie, zwemmen (id (jan)))).}
\]

By convention, the first argument of a relation corresponds to the agent. Hence, the subexpression relation in (10) expresses the semantical dependencies of sentence (8).

The dependency arrows of (8) can also be obtained geometrically. It suffices to represent the dependencies by curved overlinks.

\(\text{adapted from examples in (Bresnan et al., 1987)}\)
The vertical arrows indicate the functional symbols, connected to their arguments by the dotted overlinks.

In (11), the path starting at \( x \) and ending at \( \text{jan} \) and the path from \( x_2 \) to \( \text{marie} \) constitute the dependency links of (8). The dependence of \( \text{zien} \) on \( x_2 \) and \( z \) is indicated by dotted overlinks.  

The number of noun phrases and causal verbs is not limited, for example

Consider the entries \( \text{Eva} : \text{NP} :: \text{eva}, \text{Piet} : \text{NP} :: \text{piet}, \text{Jan} : \text{NP} :: \text{jan} \) and

\[
\begin{align*}
\text{zag} & \quad : \text{NP} \times \text{NP} \times \text{NP} \times \text{id} \times \text{id} \times \text{id} :: \text{zien}(x_3, z) \times \text{id}(x_2) \times \text{id}(x_1) \\
\text{leren} & \quad : \text{id} \times \text{id} \times \text{id} :: \text{leren}(x'_2, x'_1) \times \text{id}(x'_1) \\
\text{zwemmen} & \quad : \text{id} :: \text{zwemmen}(x)
\end{align*}
\]

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

\[
\begin{align*}
\text{Eva Piet Jan zag leren zwemmen} \\
\text{evapiet jan}(x_1, x_2, x_3, \text{zie} \times \text{id} \times \text{id})(x'_1, x'_2, \text{leren} \times \text{id})(x, \text{zwemmen}) \\
(\text{NP}) \times (\text{NP}) \times (\text{NP} \times \text{NP} \times \text{id} \times \text{id} \times \text{id} \times \text{id}) :: \text{zwemmen}(x)
\end{align*}
\]

Replace the simple types by the corresponding logical symbols and

\[\text{This vindicates Claudia Casadio’s idea that overlinks intervene in grammatical dependencies. The graph above the logical symbols in (11) represents the concatenation of the meanings of the words. It ‘lives’ in symmetric compact 2-categories, like the category of real vector spaces. The overlinks correspond to expansions in the symmetric 2-category, but not in the non-symmetric 2-category of derivations of the Pregroup Calculus. The meaning of the sentence is obtained by composing the concatenated meanings with the reduction.}\]
represent dependencies by curved overlinks

$$\text{(13)}$$

The oriented paths starting at $$x_3$$ respectively $$x'_2$$ respectively $$x$$ and terminating at eva respectively piet respectively jan constitute the dependency arrows of (12). The logical expression is

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

The graph (13) induces the labelled planar graph (14) belonging to a family of graphs relevant for dependency parsing, see (Kuhlmann and Nivre, 2006). The labels of the edges in (14) 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 (12).

$$\text{(14)}$$

The last verb in the clause may be intransitive, transitive, ditransitive etc. The arity of a verb $$v$$ 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

$$\text{zwemmen} \ (\text{swim}) : \delta \iota : \text{zwemmen}(x) \quad \text{(intrusive)}$$
$$\text{schrijven} \ (\text{write}) : \delta \iota \iota : \text{schrijven}(x_2, x_1) \quad \text{(transitive)}$$
$$\text{geven} \ (\text{give}) : \delta \iota \iota \iota : \text{geven}(x_3, x_2, x_1) \quad \text{(ditransitive)}$$

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, (15)

$$\text{zag} : [p]N^p \iota i^{[p - 1]} \delta : \text{zien}(x_p, z) \quad \text{id}(x_{p-1}) \ldots \text{id}(x_1), p \geq 2$$
$$\text{leren} : [p]N^p \iota i^{[p - 1]} \delta : \text{leren}(x_p, z) \quad \text{id}(x_{p-1}) \ldots \text{id}(x_1), p \geq 2$$

where $$x_1$$ corresponds to the first occurrence of $$N^p$$, $$x_2$$ to the second occurrence of $$N^p$$ and so on up to $$x_p$$, whereas $$z$$ corresponds to $$i'$$. 
A string of words \( w_1 \ldots w_l \) is a \( k \)-fold subordinate clause if its first word \( w_1 \) is \( \text{dat} \), 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 \( w_1 \ldots w_l \) has a canonical type assignment \( T_i \), \( 1 \leq i \leq l \), namely

\[
T_i = \begin{cases} 
\text{s } \bar{s}' & \text{if } i = 1 \\
\text{NP} & \text{if } 2 \leq i \leq k + 1 \\
[k]\text{NP } \bar{s}' \bar{i}'[k-1]\delta & \text{if } i = k + 2 \\
[2k - i + 2]\delta' \bar{i}'[2k - i + 1] \delta & \text{if } k + 2 < i < l \\
[2k - l + 2]\delta' \bar{i} & \text{if } i = l 
\end{cases}
\]

The proof that the canonical type assignment reduces to the clause type \( s \) uses induction on \( k \) and follows from the next two lemmas.

A type \( T \) is said to be \( p \)-infinitival if either \( T = [p]\delta' \bar{i} \) and \( p \geq 1 \) or \( T = [p]\delta' \bar{i}'[p - 1] \delta \) and \( p \geq 2 \). It is said to be causal if the latter holds and non-causal in the former case.

**Lemma 3.5.** Let \( T_j \) be infinitival or equal to \( \text{NP} \) for \( 1 \leq j \leq n \). Then \( T_1 \ldots T_n \not\rightarrow 1 \)

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

**Lemma 3.6.** Let \( k \geq 1 \), \( n \geq 1 \), \( Z = \bar{i}'[k]\delta \) and \( T_j \) be \( p_j \)-infinitival of length \( q_j \), \( 1 \leq j \leq n \) such that \( ZT_1 \ldots T_n \rightarrow 1 \) holds. Then

i) \( T_n \) is non-causal
ii) \( T_j \) is causal, \( j \leq n - 1 \). Moreover, \( p_j = k - j + 1 \), \( j \leq n \)
iii) \( n = k - p_n + 1 \)
iv) \( q_j = 3 + 2(k - j) \), \( 1 \leq j < n \), \( q_n = k - n + 2 \).
Proof. From $ZT_1 \ldots T_n \rightarrow 1$ follows that $T_n = [p_n] \delta^i\hat{i}$ by the same argument as above. Hence i) holds.

Next show ii), iii) and iv) by induction on $n$.

Case $n = 1$. From $\hat{i}'[k] \delta[p_1] \delta^i\hat{i} \rightarrow 1$ follows that $k = p_1$.

Case $n \geq 2$. Recall that $ZT_1 \ldots T_n = \hat{i}'[k] \delta[p_1] \delta^i\hat{i}XY \rightarrow 1$ where $Y = T_2 \ldots T_n$ and either $X = 1$ or $X = \hat{i}'[p_1-1] \delta$.

The latter alternative holds if $T_1$ is causal, the former if it is non-causal. Note that $\hat{i}'$ does not occur in the string $XY$ and therefore the leftmost occurrence of $i$ in $ZT_1 \ldots T_n \rightarrow 1$ is linked to the unique $\hat{i}'$ on its left. It follows that $[k] \delta[p_1] \delta^i \rightarrow 1$ and $XY \rightarrow 1$, hence $k = p_1$. If $X = 1$ then $T_2 \ldots T_n = Y \rightarrow 1$, contradicting Lemma 3.5. and the assumption $n \geq 2$. Hence $X = \hat{i}'[p_1-1] \delta$ 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). □

Theorem 3.1. 1) For every subordinate clause there is a unique type assignment with a reduction to the clause type $s$. 2) Every string of words from the dictionary that has a type assignment with a reduction to $s$ is a subordinate clause.

Proof. 1) The first assertion follows from the definitions by Lemmas 2.1.-2.6.

2) To see the converse, let $T_j \in D(w_j)$ for $1 \leq j \leq l$ and $T_1 \ldots T_l \rightarrow s$. By Lemma 2.5., $s$ occurs in exactly one type $T_i$ and therefore $T_i = ss^i$ and $w_i = dat$. Then $\bar{s}$ also has exactly one occurrence in the string. Indeed, each of its occurrences is linked to some occurrence of $\bar{s}^i$ and the latter occurs only together with $s$. Let $p$ be the unique index such that $\bar{s}$ 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 $\bar{s}^i$ and $\bar{s}$ are linked. Moreover, $T_p = [k]NP^r \bar{s} \hat{i}'[k-1] \delta$ for some $k \geq 2$.

First note that $i = 1$, because $T_1 \ldots T_{r-1} \rightarrow 1$ by choice of $i$. This is only possible if the string is empty by Lemma 3.5..

Next, $T_2 \ldots T_{p-1}[k]NP^r \rightarrow 1$, because $\bar{s}^i$ is linked to $\bar{s}$.

From this follows that $T_2 = \cdots = T_{p-1} = NP$ and $p = 1 + k + 1$ by Lemma 2.4.

Finally, from the preceding follows that $\hat{i}'[k-1] \delta T_{p+1} \ldots T_l \rightarrow 1$. Note that $NP$ cannot occur in this string and conclude by Lemma 3.6.

□ Theorem 3.1. above implies that for every $s$-sentence $w_1 \ldots w_l$ there are unique types $T_i \in D(w_i)$, $1 \leq i \leq l$, and a unique type $T = T_1 \ldots T_l$ such that $T \rightarrow s$. Call $T$
the canonical type, $T_i \in \mathcal{D}(w_i)$ the canonical type assignment and the unique reduction of $T$ to $s$ 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 $s$ is bounded by a polynomial. The property also intervenes in the complexity estimate of the parsing algorithm in Section 4.

**Corollary 3.2.** 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, $k \leq l/2$.

**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 3.6. The number of words before the key-word $w_p$ is $k + 1$. Hence $l = 2k - m + 2$ and therefore $k \leq l/2$.

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

$$q' = \sum_{j=1}^{n} (2j + 2m + 1) = n(2m + 1) + n(n + 1) = k^2 + 2k - m^2 - 2m$$

Hence, $q = k^2 + 3k - m^2 - m + 3 \leq k^2 + 3k + 1$. $\square$

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

**Lemma 3.7.** Let $k > n \geq 0$ and $G = \langle V_0 \cup V_1, E_0 \cup E_1 \rangle$ be the oriented graph defined as follows

- $V_0 = \{a_{ij} : 0 \leq i \leq n, 1 \leq j \leq k - i\}$ (functional symbol)
- $V_1 = \{x_{lp} : 1 \leq l \leq n + 1, 1 \leq p \leq k - l\}$ (variable)

and

- $E_0 = \{(a_{ij}, x_{i+1,j}) : 0 \leq i \leq n, 1 \leq j \leq k - i\}$ (substitution)
- $E_1 = \{(x_{il}, a_{i,l-1}) : 1 \leq i \leq n, 2 \leq l \leq k - i\}$ (identity axiom).
Then for $1 \leq j \leq k$ there is a unique maximal path starting at $a_{0j}$. Moreover, $x_{ij+1}$ and $a_{i,j-1}$ are on this path for all $l$ such that $1 \leq l \leq n + 1$ and $j - l \geq 0$ and all $i$ satisfying $j - i \geq 1$ and $0 \leq i \leq n$.

Proof. Straightforward by induction on $k$. The graph looks like this for $k = 4, n = 2$, where the underlinks represent the edges in $E_0$ and the overlinks the edges in $E_1$.

\[ (a_0 a_2 a_3 a_1) (x_1 x_3 x_2 x_1) (a_1 a_2 a_3) (x_3 x_2 x_1) (a_2 a_3) (x_2 x_1) . \]

\[ \square \]

**Pregroup Grammar with Copying Rules**

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

\[
\begin{align*}
\text{dictionary entries} & \quad \text{copying rules} \\
\text{zag} & (w_0): \mathcal{N}^p \mathcal{N}^p \mathcal{s}^i \mathcal{d}^i & \mathcal{N}^p \mathcal{T}^d \rightarrow \mathcal{N}^p \mathcal{N}^p \mathcal{T}^d \\
\text{leren} & (w_{an}): \mathcal{d}^R \mathcal{A}^R \mathcal{i}^j \mathcal{d}^i & \mathcal{d}^R \mathcal{T}^d \rightarrow \mathcal{d}^R \mathcal{R}^d \mathcal{R}^d \\
\end{align*}
\]

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 $w_1, \ldots, w_l$ from the infinite dictionary and $r$ a reduction to the clause type. Recall that $r$ corresponds to a derivation using only the contraction rule. Moreover, $T_1, \ldots, T_l$ is the canonical type assignment by Theorem 3.1. Define $T'_1 = [2] \mathcal{N}^p \mathcal{s}^i \mathcal{d}^i$ if $T_j = [p_j] \mathcal{N}^p \mathcal{s}^i \mathcal{d}^i [p_j - 1] \mathcal{d}, T'_j = [2] \mathcal{d}^R \mathcal{i}^j \mathcal{d}^i$ if $T_j = [p_j] \mathcal{d}^R \mathcal{i}^j [p_j - 1] \mathcal{d}$ and $T'_j = T_j$ else. Derive the clause type from $T'_1 \ldots T'_l$ by applying $p_j - 2$ times the
copying rule to $T'_j \neq T_j$. The resulting compound type is $T_1 \ldots T_l$.

Then apply the contraction rules 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. The argument is similar to that establishing Lemma 3.6. and Theorem 3.1.

The chosen copying rules are language specific. A general development of pregroup grammars with copying rules is beyond the scope of this paper.

**Swiss German Subordinate Clause**

According to the analysis of Shieber (1987), 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 $NP_{nom}, NP_{dat}, NP_{acc}$ as well as the dummy types $\delta_{nom}, \delta_{dat}, \delta_{acc}$. 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 Michaelis and Kracht (1997). The dictionary lists an infinite number of distinct words. Indeed, Old Georgian uses genitive suffixes for possessive compound noun phrases. The genitive suffix, denoted here $G$, is appended to noun(stem)s or names. When the construction is repeated, the previous genitive suffixes are also repeated.

```
govel-i igi sisxl-i saxl-isa-j m-is Saul-is-isa-j
all-Nom Art=Nom blood-Nom house-G-Nom Art-G Saul-G-G-Nom
'all the blood of the house of Saul'.
```

More generally, compound nominative noun phrases have the form

$$N_1{-}Nom \ N_2{-}G{-}Nom \ N_3{-}G^2{-}Nom \ldots N_k{-}G^{k-1}{-}Nom.$$ (17)

Square brackets highlight semantic dependencies as follows

$$\llbracket N_1{-}Nom \ \llbracket N_2{-}G{-}Nom \ldots \llbracket N_k{-}G^{k-1}{-}Nom \rrbracket_{NP_1} \ldots \rrbracket_{NP_l}. $$
Assume the basic types $NP_{nom}$, $N_{nom}$ and $G$ for nominative noun phrases, nominative common nouns and genitive suffixes in that order. For each $p \geq 0$, the word $Name-G^p-Nom$ respectively $Noun-G^p-Nom$ has two entries in the dictionary, namely

\[
\begin{align*}
Name-G^p-Nom : & \begin{cases} 
N_{nom}[p]G \\
N_{nom}[p]G[p+1]G' N_{nom}^G
\end{cases} \\
Noun-G^p-Nom : & \begin{cases} 
N_{nom}[p]G \\
N_{nom}[p]G[p+1]G' N_{nom}^G
\end{cases}
\end{align*}
\]

Common nouns are preceded\(^5\) by a determiner to form noun phrases like

\[(18)\]
\[
Art = Nom \quad Noun-Nom \\
Art-G \quad Noun-G^p-Nom.
\]

Adding the following entries to the dictionary

\[
\begin{align*}
Art = Nom : & NP_{nom} N_{nom}^G \\
Art-G : & NP_{nom} G N_{nom}^G
\end{align*}
\]

each of the noun phrases (18) has two possible types, each of which reduces to the type of a noun phrase. For $p \geq 1$, the types for $Art-Gen Noun-Gen^p-Nom$ are

\[
\begin{align*}
(\langle NP_{nom} G G' N_{nom}^G \rangle \langle N_{nom}[p]G \rangle) & \rightarrow NP_{nom}[p]G \\
(\langle NP_{nom} G G' N_{nom}^G \rangle \langle N_{nom}[p]G[p+1]G' N_{nom}^G \rangle) & \rightarrow
NP_{nom}[p]G[p+1]G' N_{nom}^G.
\end{align*}
\]

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 (17) 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

\(^5\)The determiner may also follow its noun. This is ignored here.
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, 2007a). 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 $L_i$ 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 $\emptyset$ 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 $\text{top}$ and $\text{pop}$ 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 set of stages $S_{w_1 \ldots w_l}$, which is union of the
following three sets

\{s_m\},
\{(i; T_1, \ldots, T_{i-1}, 1; 0) : 1 \leq i \leq l + 1, T_j \in \mathcal{D}(w_j), 1 \leq j \leq i - 1\}
and
\{(i; T_1, \ldots, T_i; p) : 1 \leq i \leq l, 1 \leq p \leq q_i, T_j \in \mathcal{D}(w_j), 1 \leq j \leq i\}

where \(q_i\) is the length of the type \(T_i\) and \(s_m\) is a new symbol denoting the initial stage.

Define a partial order on the set of stages such that \(s \leq s\) for all \(s\) and \((i; T_1, \ldots, T_i; p) \leq (i'; T_1', \ldots, T_i'; p')\)
if and only if one of the following conditions holds

\[i < i', T_j = T_j' \text{ for } 1 \leq j \leq i\]
or
\[i = i', p = 0, T_j = T_j' \text{ for } 1 \leq j < i\]
or
\[i = i', 1 \leq p \leq p', T_j = T_j' \text{ for } 1 \leq j \leq i.\]

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

\[
(i; T_1, \ldots, T_i; p) - 1 =
\begin{cases}
(i; T_1, \ldots, T_{i-1}; p-1), & \text{if } p \geq 2 \\
(i; T_1, \ldots, T_{i-1}, 1; 0) & \text{if } p = 1 \\
(i-1; T_1, \ldots, T_{i-1}; q_{i-1}) & \text{if } p = 0, i > 1 \\
s_m & \text{if } p = 0, i = 1.
\end{cases}
\]

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

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

The algorithm executes two subroutines, tagging and testing, for each word \(w_i\). When at tagging stage \((i; T_1, \ldots, T_{i-1}, 1; 0), i \leq l\), the algorithm has finished processing the type \(T_1 \ldots T_{i-1}\). The tagging
routine $\text{tag}_D$ either chooses a type $\text{Tag} \in \mathcal{D}(w_i)$ or decides to stop and updates the constant $\text{output}$. The computation of $\text{Tag}$ involves a constant $\text{key}$ that depends on the language whose sentences are to be parsed. The routine $\text{tag}_D$ 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 $s = (i; T_1, \ldots, T_l; p)$ are strings of simple types $T_j = t_{j1} \cdots t_{jq_j}$. Each testing stage $s = (i; T_1, \ldots, T_l; p)$ defines a working position $p(s) = q_1 + \cdots + q_i + p$, the simple type read $t_{p(s)} = t_p$ and the type processed $T(s) = t_1 \cdots t_{p(s)} = t_{11} \cdots t_{q_1} \cdots t_{11} \cdots t_{ip}$. To keep notation uniform, define $p(s_0) = 0$ and $t_0 = 1$. Note that for every testing stage $s$ and positive integer $i' \leq p(s)$, there is a unique testing stage $s' \leq s$ such that $i' = p(s')$.

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 $R(s)$. 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

$$I(s_m) = 1, I(S', j)) = I(S' t_j).$$

**Definition 7. Tagger-Parser**

\begin{itemize}
\item At the initial stage $s = s_0$, key, output, $S$ and $R$ are initialized to $\text{key} = \text{undefined}$, $\text{output} = \text{undefined}$, $S(s_0) = (\emptyset, 0)$, $R(s_0) = \emptyset$.
\end{itemize}

Then the process goes to the first tagging stage

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

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

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

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

$$\text{output} = \langle R(s - 1), T(s - 1), I(s - 1) \rangle$$
If $i \leq l$, the next type $T_i$ is chosen

$$tag_D(i)$$

$$T_i = Tag$$

and the process goes to the next stage unless $tag_D(i)$ updates output to fail

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

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

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

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

Then the process goes to the next stage

$$s = s + 1.$$ 

The proof of the next lemma is given in (Preller, 2007a).

**Lemma 4.1.** 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 $R(s)$ 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 $L_1$, 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.

$$tag_D(i)$$

if $i = 1$

if $l/2 \neq [l/2]$ let output $=$ fail

else $key = l/2 + 1$

if $i < key$ let $Tag = w_i$
if \( i = \text{key} \) let Tag = \( w_{r-1}^r \ldots w_i^r \bar{w}_{i-1} \ldots \bar{w}_2 \)
if \( i > \text{key} \) let Tag = \( \bar{w}_r^r \).

In the case of \( L_3 \), the routine \( \text{tag}_{D_3} \) 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 \( \text{key} = 2m + n + 1 \).

\( \text{tag}_{D_3}(i) \)

- if \( i = 1 \)
  - if \( l/2 \neq \lceil l/2 \rceil \) let \( \text{output} = \text{fail} \)
  - else let Tag = \( w_i \)
- if \( i > 1 \) and \( \text{key} = \text{undefined} \)
  - let Tag = \( w_i \)
  - if \( w_i \neq w_{i-1} \) let \( \text{key} = l/2 + i, m = i - 1, n = l/2 - (i - 1) \)
  - if \( i < \text{key} \) let Tag = \( w_i \)
- if \( i = \text{key} \) let Tag = \( [m]w_{m+n+1}^r[n]w_{m+1}^r[m]w_i^r \bar{w}_i^r[n] \bar{w}_i \)
- if \( i > \text{key} \) let Tag = \( \bar{w}_i^r \)

The case of \( L_2 \) is similar with the appropriate adaptations.

Finally, in the case of the Dutch dictionary \( D_4 \), the types are chosen according to the following properties of a subordinate clause:
- the key-index \( \text{key} \) 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.

\( \text{tag}_{D_4}(i) \)

- if \( \text{key} = \text{undefined} \)
  - if \( w_i \) is an infinitive let \( \text{output} = \text{fail} \)
  - else
    - if \( w_i \) is not a causal verb in finite from let Tag \( \in D(w_i) \)
    - if \( w_i \) is a causal verb in finite form and \( i > 1 \) let \( \text{key} = i, k = i - 1 \)
      and Tag = \( [k]NP^r \bar{s} i^r[k-1] \delta \) else let \( \text{output} = \text{fail} \)
  - if \( i > \text{key} \)
    - if \( w_i \) is an infinitive
      - of a causal verb let \( p = k - (i - \text{key}) \) and Tag = \( [p] \delta^r i^r[p-1] \delta \)
      - of a non-causal verb let Tag \( \in D(w_i) \)
    - else \( \text{output} = \text{fail} \).

When fed the string of words Marie Jan zag zwemmen the parser goes through the following stages and values. The constant \( \text{key} \) re-
mains *undefined* until a causal verb in finite form is encountered. Hence, the constants $Tag, S, \mathcal{R}$ change like this

<table>
<thead>
<tr>
<th>$s_m$</th>
<th>$Tag$</th>
<th>$S$</th>
<th>$\mathcal{R}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1;1;0</td>
<td>$T_1 = NP$</td>
<td>$\langle \emptyset, 0 \rangle$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>1;1;1</td>
<td>$NP$</td>
<td>$\langle \langle \langle \emptyset, 0 \rangle, 1 \rangle \rangle$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2;1;1</td>
<td>$NP$</td>
<td>$\langle \langle \langle \langle \emptyset, 0 \rangle, 1 \rangle \rangle \rangle$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

At tagging stage $(3; T_1, T_2, 1; 0)$, the value of $key$ is updated to 3, because $w_3$ is the first causal verb in finite form. Moreover, the tag is updated to $Tag = T_3 = NPNP \hat{s}i^i \delta$ and remains unchanged till the next tagging stage. The values of $S, \mathcal{R}$ are updated as follows

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\mathcal{R}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(3; T_1, T_2, T_3; 1)$</td>
<td>$\langle \langle \langle \langle \emptyset, 0 \rangle, 1 \rangle \rangle \rangle$</td>
</tr>
<tr>
<td>$(3; T_1, T_2, T_3; 2)$</td>
<td>$\langle \emptyset, 0 \rangle$</td>
</tr>
<tr>
<td>$(3; T_1, T_2, T_3; 3)$</td>
<td>$\langle \langle \langle \langle \emptyset, 0 \rangle, 5 \rangle \rangle \rangle$</td>
</tr>
<tr>
<td>$(3; T_1, T_2, T_3; 4)$</td>
<td>$\langle \langle \langle \langle \emptyset, 0 \rangle, 5 \rangle \rangle \rangle \langle \emptyset, 6 \rangle$</td>
</tr>
<tr>
<td>$(3; T_1, T_2, T_3; 5)$</td>
<td>$\langle \langle \langle \langle \langle \emptyset, 0 \rangle, 5 \rangle \rangle \rangle \langle \emptyset, 6 \rangle \langle \emptyset, 7 \rangle$</td>
</tr>
</tbody>
</table>

At tagging stage $(4; T_1, T_2, T_3, 1; 0)$, the tag is updated to $Tag = T_4 = \delta^i \hat{i}$. It remains unchanged till the maximal stage

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\mathcal{R}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(4; T_1, \ldots, T_4; 1)$</td>
<td>$\langle \langle \langle \langle \langle \emptyset, 0 \rangle, 5 \rangle \rangle \rangle \langle \emptyset, 6 \rangle \langle \emptyset, 7 \rangle \rangle \rangle \langle \emptyset, 8 \rangle$</td>
</tr>
<tr>
<td>$(4; T_1, \ldots, T_4; 2)$</td>
<td>$\langle \emptyset, 0 \rangle \langle \emptyset, 5 \rangle \langle \emptyset, 6 \rangle \langle \emptyset, 7 \rangle \langle \emptyset, 8 \rangle \langle \emptyset, 9 \rangle$</td>
</tr>
</tbody>
</table>

At the maximal stage $s_{fm} = (5; T_1, \ldots, T_4, 1; 0)$, the *output* is updated to

- $\mathcal{R}(s_{fm}) = \{[2,3], [1,4], [7,8], [6,9]\}$
- $I(s_{fm}) = NPNPNP \hat{s}i^i \delta^i \hat{i}$
- $I(s_{fm}) = \emptyset$.

One could argue following Lambek (2008b) that simple types represent bits of information to be stored in the short-term memory (limited to 7±2 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 \([p]p^{\prime \ast} [p-1]0\), 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 4.1. 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 \(output = \langle R, T, s \rangle\). If this is the case, \(R\) is a reduction of \(T\) to \(s\). Moreover, the algorithm is linear for the formal languages and square polynomial for the natural languages.

Proof. The first assertion follows from Lemma 4.1. and Lemma 2.1. 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 \(T(s_{fin})\).

\[\square\]

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, see (Pentus, 2003). 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, 2008) show that such algorithms are highly accurate for other formalisms in general where no proof of completeness exists.

Acknowledgement

Compact bilinear logic, one of Joachim Lambek’s more recent inventions, is a compactification of higher order logic to second order logic. It is also a simple mathematical tool for many fascinating topics in computational linguistics. Proofs are represented by planar graphs. The pregroup grammars, which are based on this logic, have polynomial parsing algorithms, even for context sensitive language fragments. They allow semantical interpretation reflecting the dependency links. The grammars are also suited for handling large amounts of data, modelling dialog and language learning. I’m grateful for the occasion this Festschrift offers me to thank Joachim Lambek for his beautiful and powerful invention.

My thanks go also to an anonymous reader of the present paper for many helpful remarks.

Works Cited


