

Polynomial Pregroup Grammars parse Context Sensitive Languages

ANNE PRELLER
*LIRMM/CNRS, Montpellier, France*¹

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’.

¹I 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 $T_1 \dots T_l$ associated to $w_1 \dots w_l$ 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 $P(\mathcal{B})$ generated by a partially ordered set $\mathcal{B} = \langle B, \leq \rangle$ is the free monoid generated by the set of *simple types*

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

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

an equality $T = t_1 \dots t_n$, it is always understood that the lower case t_i 's are simple types. In the case where $n = 0$, the string $t_1 \dots t_n$ 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

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

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

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

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 (2002).

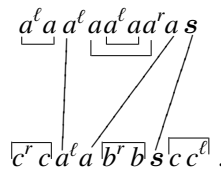
Definition 1. (Dictionaries) Let Σ be a non-empty set. A *pregroup dictionary* for Σ based on \mathcal{B} is a map \mathcal{D} defined on Σ with values in the set of subsets of $\mathcal{P}(\mathcal{B})$. A *type assignment* of $w_1 \dots w_n$ is a sequence of types T_1, \dots, T_n for which $T_i \in \mathcal{D}(w_i)$, $1 \leq i \leq n$. 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 Σ 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 Σ^*

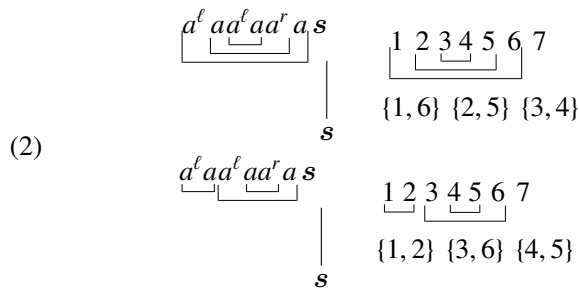
$$\mathcal{L}_{\mathcal{G}} = \{w_1 \dots w_n : T_1 \dots T_n \rightarrow s \text{ for some } T_i \in \mathcal{D}(w_i), 1 \leq i \leq n\}.$$

By definition, $T \rightarrow T'$ if and only if there is a sequence of types T_1, \dots, 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 $a^l a a^l a a^l a^r a s$ to the type $c^r a^l a b^r b s c c^l$



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

- 1) for $i \in \{1, \dots, n\} - \{i_1, \dots, i_k\}$ there is exactly one $j \leq n$ such that $\{i, j\} \in R$
- 2) if $\{i, j\} \in R$ then $i \neq j$, $i \notin \{i_1, \dots, i_k\}$, $j \notin \{i_1, \dots, i_k\}$
- 3) for $\{i, j\} \in R$ and $i < l < j$ then there is m such that $i < m < j$ and $\{l, m\} \in R$
- 4) $t_i t_j \rightarrow 1$ for $i < j$ and $\{i, j\} \in R$.

The vertices in $\{1, \dots, n\}$ 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 $T = t_1 \dots t_n$ 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^ℓ and s occur in sa^ℓ , 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 $T = t_1 \dots t_n$ is irreducible if $t_i t_{i+1} \not\rightarrow 1$ 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 $aa^\ell b' b$ 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, $ba^\ell aa^r a$ has the two irreducible forms b and $ba^r a$, 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 $T = t_1 \dots t_m$ and suppose that $t_i \dots t_j \rightarrow 1$ for some $1 \leq i < j \leq m$. Then $t_1 \dots t_{i-1} t_{j+1} \dots t_m \rightarrow T'$.*

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

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

$$t_1 \dots t_i S t_{i+1} \dots t_m \rightarrow T'.$$

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 $t_i t_j$ is a *block* of $T = t_1 \dots t_m$ if $i < j$ and t_i is immediately followed by t_j in every irreducible form of T .

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

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

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

Eva : NP :: eva
promised : $NP^r s \delta \bar{i}^\ell NP^\ell$:: $\text{promise}(x_1, x_2, x_3) \text{ id}(x_1)$
asked : $NP^r s \delta \bar{i}^\ell NP^\ell$:: $\text{ask}(x_1, x_2, x_3) \text{ id}(x_3)$
Jan : NP :: jan
to : \bar{i}^ℓ :: $\text{to}(y)$
come : $i\delta^r$:: $\text{come}(z)$

The basic types NP , i and \bar{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,

$$\begin{array}{ccccccc} NP^r & s & \delta & \bar{i}^\ell & NP^\ell & & \\ x_1 & \text{ask} & \text{id} & x_2 & x_3 & \cdot & \end{array}$$

The variables on which a logical expression depends render the intuitive meaning of semantical dependency. The translations of

$promised : NP^r s \bar{\delta} \bar{i}^\ell NP^\ell$ and $asked : NP^r s \delta \bar{i}^\ell NP^\ell$ differ by the variable on which the translation id of the basic type δ 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

$$\begin{array}{c} Eva \quad promised \quad Jan \quad to \quad come \\ \underbrace{(NP)} \quad \underbrace{(NP^r \quad s \quad \bar{\delta} \quad \bar{i}^\ell \quad NP^\ell)} \quad \underbrace{(NP)(\bar{i} \quad \bar{i}^\ell)(i \quad \delta^r)} \end{array}$$

do the following

- write the corresponding logical symbols above the simple types²

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

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

$$\begin{array}{c} \underbrace{eva \quad x_1} \quad \underbrace{promise} \quad \underbrace{id \quad x_2 \quad x_3 \quad jan \quad to \quad y \quad come \quad z} \end{array}$$

- define the substitutions according to the links

(3)

$$x_1 \mapsto eva \quad x_2 \mapsto to(y) \quad x_3 \mapsto jan \quad y \mapsto come(z) \quad z \mapsto id(x_1)$$

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

$$promise(eva, to(come(id(eva))), jan)$$

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

$$id(x) = x$$

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

$$(4) \quad promise(eva, to(come(eva)), jan)$$

²Recall: 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

$$\begin{array}{c}
 \text{Eva} \quad \text{asked} \quad \text{Jan to come} \\
 \underbrace{(\text{NP})} \underbrace{(\text{NP}^r \text{ s } \delta \bar{i}^\ell \text{ NP}^\ell)} \underbrace{(\text{NP})(\bar{i} i^\ell)(i \delta^r)},
 \end{array}$$

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

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

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 $T_1 \dots T_l$ is $O(l^n)$ for every every type assignment $w_1 : T_1, \dots, w_l : T_l$ for which $T_1 \dots T_l \rightarrow s$.

If the length of types occurring in the dictionary does not exceed a constant α , then the corresponding grammar is *linear* polynomial, i.e. of degree 1. Indeed, for every string of words $w_1 \dots w_l$ the length of the assigned type $T_1 \dots T_l$ is bounded by the αl . 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

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

3.1.1. Duplication

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

The set of basic types is constructed from Σ 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 Σ . The set of basic types $\mathcal{B}_1 = \langle B_1, = \rangle$ is ordered by equality, where

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

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

$$\mathcal{D}_1(a) = \{a\} \cup \{\bar{a}^r\} \cup \{b_1^r \dots b_i^r a^r s \bar{b}_1 \dots \bar{b}_i : b_1 \dots b_i \in \Sigma^*\}.$$

This dictionary is modest (recall Definition 4).

Lemma 3.1. *The language $\mathcal{L}_{\mathcal{G}_1}$ of the pregroup grammar $\mathcal{G}_1 = \langle \mathcal{D}_1, s \rangle$ contains \mathcal{L}_1 .*

Proof. Assume that $X = a_1 \dots a_m \in \mathcal{L}_1$. 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 $T_2 = a_1^r s$. 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

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

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

The type assignment T_1, \dots, T_m defined above is called the *canonical type assignment* and the concatenated type $T_1 \dots T_m$ 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 $T_j \in \mathcal{D}_1(a_j)$, $1 \leq j \leq m$, is a type assignment for $X = a_1 \dots a_m \in \Sigma^*$ such that $T_1 \dots T_m \rightarrow s$. Then $X \in \mathcal{L}_1$ and $T_1 \dots T_m$ is the canonical type assignment for X .*

Proof. Two things are to be proved: $m = 2n$ for some $n \geq 1$ and that $a_{n+i} = a_i$ for $1 \leq i \leq n$. The assumption $T = T_1 \dots T_m \rightarrow s$ 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

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

for some $i \in \mathbb{N}$ and some string $b_1 \dots b_i \in \Sigma^*$. Moreover, if $j \neq k$ then $T_j = a_j$ or $T_j = \bar{a}_j^r$. From Lemma 2.6. follows that

$$\begin{aligned} \bar{b}_1 \dots \bar{b}_i T_{k+1} \dots T_m &\rightarrow 1 \\ T_1 \dots T_{k-1} b_1^r \dots b_i^r a_k^r &\rightarrow 1. \end{aligned}$$

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

$$(6) \quad \begin{aligned} T_{k+j} &= \bar{a}_{k+j}^r \text{ for } 1 \leq j \leq i \\ a_{k+j} &= b_{i-j+1} \text{ for } 1 \leq j \leq i \\ i &= m - k \geq 0. \end{aligned}$$

Case $i = 0$:

Then $T_k = a_k^r s$ and $T_{k+1} \dots T_m \rightarrow 1$, by Lemma 2.6.. 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 $T_1 \dots T_m$ by Lemma 2.4., contradicting the assumption. For the same reason, $\bar{b}_i \bar{a}_{k+1}^r \rightarrow 1$, i.e. $\bar{b}_i = \bar{a}_{k+1}$. The latter implies $a_{k+1} = b_i$. Let $T' = \bar{b}_1 \dots \bar{b}_{i-1} T_{k+2} \dots T_m$. Then $T' \rightarrow 1$ by Lemma 2.2.. 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 $T_1 \dots T_{k-1} b_1^r \dots b_i^r a_k^r \rightarrow 1$ show that

$$(7) \quad \begin{aligned} T_{k-l} &= a_{k-l} \text{ for } 1 \leq l \leq i+1 \\ a_{k-l} &= b_l \text{ for } 1 \leq l \leq i \\ a_{k-i-1} &= a_k \\ 1 &= k - i - 1 \end{aligned}$$

by induction on i . In the case $i = 0$, note that $k > 1$, because if $k = 1$ then $a_k^r \rightarrow 1$, which is impossible. It follows that $T_{k-1} = a_{k-1}$ and $a_{k-1} = a_k$. Hence $T_1 \dots T_{k-2} \rightarrow 1$ 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 $a_{j+1} = a_{2+j-1} = a_{k-i+j-1} = b_{i-j+1} = a_{k+j} = a_{n+j+1}$, for $1 \leq j \leq i = n-1$. Finally, $a_1 = a_{n+1}$ follows from $a_{k-i-1} = a_k$. \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

$$\begin{array}{ccccccccccc} a_1 & a_2 & \dots & a_{k-1} & & a_k & & a_{k+1} & \dots & a_m \\ a_1 & a_2 & \dots & a_{k-1} & (b_1^r \dots b_i^r a_k^r) & s & \bar{b}_1 \dots \bar{b}_i & \bar{a}_{k+1}^r & \dots & \bar{a}_m^r \end{array}$$

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 $a^{(m)}$ 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

$$\mathcal{B}_2 = \{s\} \cup \Sigma \cup \{\bar{a} : a \in \Sigma\}$$

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

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

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

Lemma 3.3. *A string $X = a_1 \dots a_m \in \Sigma^*$ has a type assignment $a_i : T_i \in \mathcal{D}_2$, $1 \leq i \leq m$, such that $T_1 \dots T_m \rightarrow \mathbf{s}$ if and only if $X \in \mathcal{L}_2$.*

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

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

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 $T_1 \dots T_m \rightarrow \mathbf{s}$. The argument is similar to that of Lemma 3.2.. Now the unique type with an occurrence of the sentence type has the form

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

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

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

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

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

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

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

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

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

where $a \neq b$, $b \neq c$, $c \neq d$. Again, \mathcal{D}_3 is modest and $\mathcal{G}_3 = \langle \mathcal{D}_3, \mathbf{s} \rangle$ generates \mathcal{L}_3 .

Lemma 3.4. $X = a_1 \dots a_l \in \Sigma^*$ has a type assignment $T_i \in \mathcal{D}_3(a_i)$, $1 \leq i \leq l$, such that $T_1 \dots T_l \rightarrow \mathbf{s}$ if and only if $X \in \mathcal{L}_3$.

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

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

$$T_k = [m]a_{m+n+1}^r [n]a_{m+1}^r [m]a_1^r s \bar{a}_k^\ell [n]\bar{a}_k.$$

The details are left to the reader. \square

Note the common features shared by the three grammars \mathcal{G}_1 - \mathcal{G}_3 . Sentences $w_1 \dots w_l$ have a canonical type assignment T_1, \dots, 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 $t_1 \dots t_q = T_1 \dots T_l$ 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 clause³ 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.

$$(8) \quad \begin{array}{c} \text{dat Marie Jan zag zwemmen} \\ \text{(that Mary saw Jan swim)} \end{array}$$

Using the entries

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

parse this clause

$$(9) \quad \begin{array}{c} \text{dat Marie Jan zag zwemmen} \\ \text{dat marie jan , (x}_1 \text{ x}_2 \text{ zien z id) (x zwemmen)} \\ \underbrace{s \bar{s}^\ell (NP) (NP) (NP^r NP^r \bar{s} \bar{i}^\ell \delta)}_{\text{zien}} \underbrace{(\delta^r \bar{i})}_{\text{zwemmen}} \end{array}$$

and compute its logical interpretation according to Section 2.2

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

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

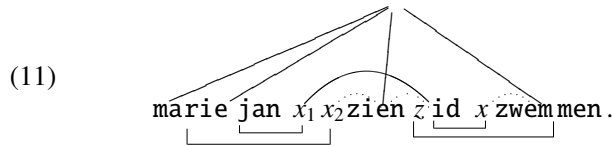
$$(10) \quad \text{dat}(\text{zien}(\text{marie}, \text{zwemmen}(\text{jan}))).$$

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 overlins.

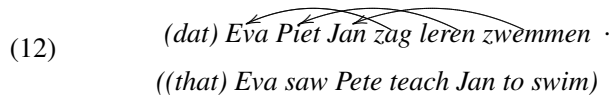
³adapted from examples in (Bresnan *et al.*, 1987)

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 jan and the path from x_2 to marie constitute the dependency links of (8). The dependence of zien on x_2 and z is indicated by dotted overlinks.⁴

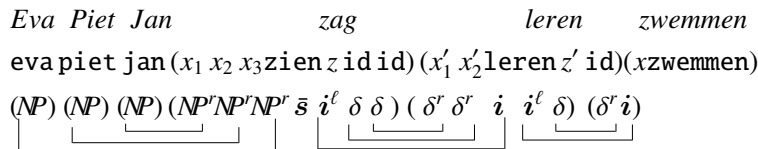
The number of noun phrases and causal verbs is not limited, for example



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

zag	$(\text{saw}) : NP^r NP^r NP^r \bar{s} i^\ell \delta \delta :: \text{zien}(x_3, z) \text{id}(x_2) \text{id}(x_1)$
leren	$(\text{teach}) : \delta^r \delta^r i^\ell \delta :: \text{leren}(x'_2, z') \text{id}(x'_1)$
zwemmen	$(\text{swim}) : \delta^r i :: \text{zwemmen}(x)$

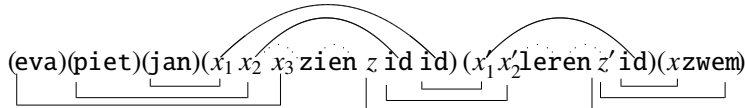
Compute the reduction of the assigned type to \bar{s}



Replace the simple types by the corresponding logical symbols and

⁴This 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 overlincs
(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

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

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 overlincs ‘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).



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

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

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,
(15)

$$\begin{array}{l} \text{zag} : [p]NP^r \bar{s}i^\ell [p-1]\delta :: \text{zien}(x_p, z) \quad \text{id}(x_{p-1}) \dots \text{id}(x_1), p \geq 2 \\ \text{leren} : [p]\delta^r i^\ell [p-1]\delta :: \text{leren}(x_p, z) \quad \text{id}(x_{p-1}) \dots \text{id}(x_1), p \geq 2 \end{array}$$

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 i^ℓ .

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

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

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

Lemma 3.5. *Let T_j be infinitival or equal to NP for $1 \leq j \leq n$. Then $T_1 \dots T_n \not\rightarrow 1$*

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

Lemma 3.6. *Let $k \geq 1$, $n \geq 1$, $Z = \bar{i}^\ell [k] \delta$ and T_j be p_j -infinitival of length q_j , $1 \leq j \leq n$ such that $ZT_1 \dots T_n \rightarrow 1$ 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 $ZT_1 \dots T_n \rightarrow 1$ follows that $T_n = [p_n]\delta^r i$ by the same argument as above. Hence i) holds.

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

Case $n = 1$. From $i^\ell [k]\delta [p_1]\delta^r i \rightarrow 1$ follows that $k = p_1$

Case $n \geq 2$. Recall that $ZT_1 \dots T_n = i^\ell [k]\delta [p_1]\delta^r iXY \rightarrow 1$ where $Y = T_2 \dots T_n$ and either $X = 1$ or $X = i^\ell [p_1 - 1]\delta$. The latter alternative holds if T_1 is causal, the former if it is non-causal. Note that i^r does not occur in the string XY and therefore the leftmost occurrence of i in $ZT_1 \dots T_n \rightarrow 1$ is linked to the unique i^ℓ on its left. It follows that $[k]\delta [p_1]\delta^r \rightarrow 1$ and $XY \rightarrow 1$, hence $k = p_1$. If $X = 1$ then $T_2 \dots T_n = Y \rightarrow 1$, contradicting Lemma 3.5. and the assumption $n \geq 2$. Hence $X = i^\ell [p_1 - 1]\delta$ with $p_1 \geq 2$. Now apply the induction hypothesis to X, T_2, \dots, T_n .

Finally, iii) and iv) are immediate consequences of ii) and i). \square

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 $T_j \in \mathcal{D}(w_j)$ for $1 \leq j \leq l$ and $T_1 \dots T_l \rightarrow s$. By Lemma 2.5., s occurs in exactly one type T_i and therefore $T_i = s\bar{s}^\ell$ and $w_i = \text{dat}$. Then \bar{s} also has exactly one occurrence in the string. Indeed, each of its occurrences is linked to some occurrence of \bar{s}^ℓ 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 \mathcal{NP} for all j other than i and p . Note that $p > i$, because \bar{s}^ℓ and \bar{s} are linked. Moreover, $T_p = [k]\mathcal{NP}^r \bar{s} i^\ell [k - 1]\delta$ for some $k \geq 2$.

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

Next, $T_2 \dots T_{p-1} [k]\mathcal{NP}^r \rightarrow 1$, because \bar{s}^ℓ is linked to \bar{s} . From this follows that $T_2 = \dots = T_{p-1} = \mathcal{NP}$ and $p = 1 + k + 1$ by Lemma 2.4..

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

\square Theorem 3.1. above implies that for every s -sentence $w_1 \dots w_l$ there are unique types $T_i \in \mathcal{D}(w_i)$, $1 \leq i \leq l$, and a unique type $T = T_1 \dots T_l$ 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 $w_1 \dots w_l$.

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 $T_1 \dots T_l$ 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 $T_1 \dots T_l$ is

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

where q' is the length of the type $T_p T_{p+1} \dots T_{l-1}$. 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

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

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

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

Lemma 3.7. *Let $k > n \geq 0$ 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*

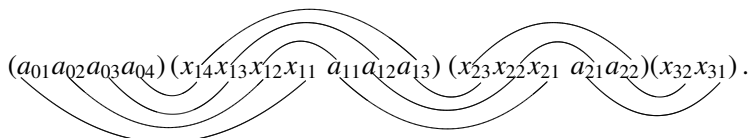
$$\begin{aligned} \mathcal{V}_0 &= \{a_{ij} : 0 \leq i \leq n, 1 \leq j \leq k - i\} && \text{(functional symbol)} \\ \mathcal{V}_1 &= \{x_{lp} : 1 \leq l \leq n + 1, 1 \leq p \leq k - l\} && \text{(variable)} \end{aligned}$$

and

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

Then for $1 \leq j \leq k$ 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 $1 \leq l \leq n+1$ 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 \mathcal{E}_0 and the overlinks the edges in \mathcal{E}_1



□

Pregroup Grammar with Copying Rules

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

	dictionary entries	copying rules
(16)	zag <small>(_{saw})</small> : $NP^r NP^r \bar{s}i^\ell \delta$ $leren$ <small>(_{teach})</small> : $\delta^r \delta^r i i^\ell \delta$	$NP^r T \delta \rightarrow NP^r NP^r T \delta \delta$ $\delta^r T \delta \rightarrow \delta^r \delta^r T \delta \delta$

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, \dots, T_n be a type assignment for w_1, \dots, 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, \dots, T_l is the canonical type assignment by Theorem 3.1.. Define $T'_j = [2]NP^r \bar{s}i^\ell \delta$ if $T_j = [p_j]NP^r \bar{s}i^\ell [p_j - 1]\delta$, $T'_j = [2]\delta^r i i^\ell \delta$ if $T_j = [p_j]\delta^r i i^\ell [p_j - 1]\delta$ and $T'_j = T_j$ else. Derive the clause type from $T'_1 \dots T'_l$ by applying $p_j - 2$ times the

copying rule to $T'_j \neq T_j$. The resulting compound type is $T_1 \dots T_l$. 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 δ_{nom} , δ_{dat} , δ_{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

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

Square brackets highlight semantic dependencies as follows

$$\llbracket N_1 - \text{Nom} \llbracket N_2 - \text{G} - \text{Nom} \ \dots \llbracket N_k - \text{G}^{k-1} - \text{Nom} \rrbracket_{NP_k} \ \dots \rrbracket_{NP_2} \rrbracket_{NP_1} .$$

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

$$\begin{aligned} \textit{Name-G}^p\text{-Nom} &: \begin{cases} NP_{\text{nom}} [p]G \\ NP_{\text{nom}} [p]G [p+1]G^\ell NP_{\text{nom}}^\ell \end{cases} \\ \textit{Noun-G}^p\text{-Nom} &: \begin{cases} N_{\text{nom}} [p]G \\ N_{\text{nom}} [p]G [p+1]G^\ell NP_{\text{nom}}^\ell \end{cases} \end{aligned}$$

Common nouns are preceded⁵ by a determiner to form noun phrases like

$$(18) \quad \begin{array}{ll} \textit{Art} = \textit{Nom} & \textit{Noun-Nom} \\ & \textit{Art-G} \quad \textit{Noun-G}^p\text{-Nom} \end{array}$$

Adding the following entries to the dictionary

$$\begin{array}{ll} \textit{Art} = \textit{Nom} &: NP_{\text{nom}} N_{\text{nom}}^\ell \\ \textit{Art-G} &: NP_{\text{nom}} GG^\ell N_{\text{nom}}^\ell \end{array}$$

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

$$\begin{aligned} (NP_{\text{nom}} GG^\ell N_{\text{nom}}^\ell) (N_{\text{nom}} [p]G) &\rightarrow NP_{\text{nom}} [p]G \\ \text{and} & \\ (NP_{\text{nom}} GG^\ell N_{\text{nom}}^\ell) (N_{\text{nom}} [p]G [p+1]G^\ell NP_{\text{nom}}^\ell) &\rightarrow \\ NP_{\text{nom}} [p]G [p+1]G^\ell NP_{\text{nom}}^\ell & \end{aligned}$$

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

⁵The 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* $w_1 \dots w_l \in \Sigma^*$ *has a type assignment* T_1, \dots, T_l *such that the concatenated type* $T_1 \dots T_l$ *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 \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, \dots, T_i that cannot be extended to a type assignment $T_1, \dots, T_i, T_{i+1}, \dots, T_l$ 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 *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 $w_1 \dots w_l \in \Sigma^*$, the algorithm moves through a subset of the set of *stages* $\mathcal{S}_{w_1 \dots w_l}$, which is union of the

following three sets

$$\begin{aligned} & \{s_{in}\}, \\ & \{(i; T_1, \dots, T_{i-1}, 1; 0) : 1 \leq i \leq l+1, T_j \in \mathcal{D}(w_j), 1 \leq j \leq i-1\} \\ & \text{and} \\ & \{(i; T_1, \dots, T_i; p) : 1 \leq i \leq l, 1 \leq p \leq q_i, T_j \in \mathcal{D}(w_j), 1 \leq j \leq i\} \end{aligned}$$

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

Define a partial order on the set of stages such that $s_{in} \leq s$ for all s and

$$(i; T_1, \dots, T_i; p) \leq (i'; T'_1, \dots, T'_{i'}; p')$$

if and only if one of the following conditions holds

$$\begin{aligned} & i < i', T_j = T'_j \text{ for } 1 \leq j \leq i \\ & \text{or} \\ & i = i', p = 0, T_j = T'_j \text{ for } 1 \leq j < i \\ & \text{or} \\ & i = i', 1 \leq p \leq p', T_j = T'_j, 1 \leq j \leq i. \end{aligned}$$

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 $(l+1; T_1, \dots, T_l, 1; 0)$ are maximal.

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

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

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

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

The algorithm executes two subroutines, *tagging* and *testing*, for each word w_i . When at tagging stage $(i; T_1, \dots, T_{i-1}, 1; 0)$, $i \leq l$, the algorithm has finished processing the type $T_1 \dots T_{i-1}$. 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 $(l + 1; T_1, \dots, T_l; 0)$, the output is updated to the result computed so far.

Recall that the types T_j at stage $s = (i; T_1, \dots, T_i; p)$ are strings of simple types $T_j = t_{j1} \dots t_{jq_j}$. Each testing stage

$$s = (i; T_1, \dots, T_i; p)$$

defines a *working position* $p(s) = q_1 + \dots + q_i + p$, the simple type *read* $t_{p(s)} = t_{ip}$ and the type *processed* $T(s) = t_1 \dots t_{p(s)} = t_{11} \dots t_{1q_1} \dots t_{i1} \dots t_{ip}$. To keep notation uniform, define $p(s_{in}) = 0$ and $t_0 = 1$. Note that for every testing stage s and positive integer $i' \leq p(s)$, 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 $\mathcal{R}(s)$. 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 $key = \text{undefined}$, $output = \text{undefined}$, $S(s_{in}) = \langle \emptyset, 0 \rangle$, $\mathcal{R}(s_{in}) = \emptyset$.

Then the process goes to the first tagging stage

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

▼ At tagging stage $s = (i; T_1, \dots, T_{i-1}, 1; 0)$, 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

$$\begin{aligned} & \text{tag}_{\mathcal{D}}(i) \\ & T_i = \text{Tag} \end{aligned}$$

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

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

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

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

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

Then the process goes to the next stage

$$s = s + 1.$$

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

Lemma 4.1. *For every stage $s = (i; T_1, \dots, T_i; p)$, 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 $w_1 \dots w_l$ 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.

$\text{tag}_{\mathcal{D}_1}(i)$
 if $i = 1$
 if $l/2 \neq \lceil l/2 \rceil$ let $\text{output} = \text{fail}$
 else $\text{key} = l/2 + 1$
 if $i < \text{key}$ let $\text{Tag} = w_i$

if $i = key$ let $Tag = w_{i-1}^r \dots w_2^r w_i^r s \bar{w}_{i-1} \dots \bar{w}_2$
 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 $w_1 \dots w_l$ is not a sentence. Otherwise, it computes the number m of repetitions of the first word w_1 . If $w_1 \dots w_l$ 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 $Tag = [m]w_{m+n+1}^r [n]w_{m+1}^r [m]w_1^r s \bar{w}_i^r [n]\bar{w}_i$
 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 form let $Tag \in \mathcal{D}(w_i)$
 if w_i is a causal verb in finite form and $i > 1$ let $key = i$, $k = i - 1$
 and $Tag = [k]NP^r \bar{s} i^l [k - 1]\delta$ else let $output=fail$
 if $i > key$
 if w_i is an infinitive
 of a causal verb let $p = k - (i - key)$ and $Tag = [p]\delta^r i i^l [p - 1]\delta$
 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 re-

mains *undefined* until a causal verb in finite form is encountered. Hence, the constants Tag, S, \mathcal{R} change like this

	Tag	S	\mathcal{R}
s_{in}	<i>undefined</i>	$\langle \emptyset, 0 \rangle$	\emptyset
$(1; 1; 0)$	$T_1 = NP$	$\langle \emptyset, 0 \rangle$	\emptyset
$(1; T_1; 1)$	NP	$\langle \langle \emptyset, 0 \rangle, 1 \rangle$	\emptyset
$(2; T_1, 1; 0)$	$T_2 = NP$	$\langle \langle \emptyset, 0 \rangle, 1 \rangle$	\emptyset
$(2; T_1, T_2; 1)$	NP	$\langle \langle \langle \emptyset, 0 \rangle, 1 \rangle, 2 \rangle$	\emptyset .

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 $Tag = T_3 = NP^r NP^r \bar{s}i^\ell \delta$ and remains unchanged till the next tagging stage. The values of S, \mathcal{R} are updated as follows

	S	\mathcal{R}
$(3; T_1, T_2, T_3; 1)$	$\langle \langle \emptyset, 0 \rangle, 1 \rangle$	$\{2, 3\}$
$(3; T_1, T_2, T_3; 2)$	$\langle \emptyset, 0 \rangle$	$\{\{2, 3\}, \{1, 4\}\}$
$(3; T_1, T_2, T_3; 3)$	$\langle \langle \emptyset, 0 \rangle, 5 \rangle$	$\{\{2, 3\}, \{1, 4\}\}$
$(3; T_1, T_2, T_3; 4)$	$\langle \langle \langle \emptyset, 0 \rangle, 5 \rangle, 6 \rangle$	$\{\{2, 3\}, \{1, 4\}\}$
$(3; T_1, T_2, T_3; 5)$	$\langle \langle \langle \langle \emptyset, 0 \rangle, 5 \rangle, 6 \rangle, 7 \rangle$	$\{\{2, 3\}, \{1, 4\}\}$.

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

	S	\mathcal{R}
$(4; T_1, \dots, T_4; 1)$	$\langle \langle \langle \emptyset, 0 \rangle, 5 \rangle, 6 \rangle$	$\{\{2, 3\}, \{1, 4\}, \{7, 8\}\}$
$(4; T_1, \dots, T_4; 2)$	$\langle \langle \emptyset, 0 \rangle, 5 \rangle$	$\{\{2, 3\}, \{1, 4\}, \{7, 8\}, \{6, 9\}\}$

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

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

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 $[p]NP^r si^\ell [p-1]\delta$, 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 4.1. *The string of words $w_1 \dots w_l$ is a sentence if and only if the tagging-parsing algorithm reaches a maximal stage such that output = $\langle \mathcal{R}, T, s \rangle$. If this is the case, \mathcal{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

1. Bhatt, Rajesh and Joshi Aravind. 2004. Semilinearity is a syntactic invariant: A reply to Michaelis and Kracht 1997. *Linguistic Inquiry*.
2. Bresnan, Joan, Ronald M. Kaplan, Stanley Peters, and Annie Zaenen. 1987. *The Formal Complexity of Natural Language*, chap. Cross-Serial Dependencies in Dutch, 286–319. *Studies in Linguistics and Philosophy*. Dordrecht: Reidel Publishing Company.

3. Buszkowski, Wojciech. 2001. Lambek Grammars based on pregroups. In *Logical Aspects of Computational Linguistics*, edited by P. de Groote, no. 2099 in LNAI. Springer.
4. —. 2002. Cut elimination for the Lambek calculus of adjoints. In *Papers in formal linguistics and logic*, edited by Michele Abrusci. Bologna, Italy: Bulzoni.
5. Francez, Nissim and Michael Kaminski. 2008. Commutation Augmented Pregroup Grammars and Mildly Context Sensitive Languages. *Studia Logica* 87(2:3): 295–321.
6. Kracht, Marcus. 2007. Compositionality: The Very Idea. *Research in Language and Computation* 5: 287–308.
7. Kuhlmann, Marco and Joakim Nivre. 2006. Midly Non-Projective Dependency Structure. In *Proceedings of the COLING/ACL on Main Conference Poster Sessions*, 507–514.
8. Lambek, Joachim. 1958. The mathematics of sentence structure. *American Mathematical Monthly* 65: 154–170.
9. —. 1999. Type Grammar revisited. In *Logical Aspects of Computational Linguistics*, edited by Alain Lecomte, LNAI, vol. 1582, 1–27. Heidelberg: Springer.
10. —. 2008a. *From word to sentence*. Milano, Italia: Polimetrica.
11. —. 2008b. Reflections on English Pronouns. In *Computational and Algebraic Approaches to Natural Language*, edited by Claudia Casadio and Joachim Lambek, 233–253. Milano, Italia.
12. Michaelis, Jens and Marcus Kracht. 1997. Semilinearity as a syntactic invariant. In *Logical Aspects of Computational Linguistics*, edited by Christian Retoré, *Lecture Notes in Computer Science*, vol. 1328, 329–345. Heidelberg: Springer.
13. Nivre, Joakim. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics* 34(4): 513–553.
14. Pentus, Mati. 2003. *Lambek calculus is NP-complete*. Tech. rep., CUNY Graduate Center, New York.
15. Pollard, Carl and Ivan A. Sag. 1994. *Head-driven phrase structure grammar*. Studies in Contemporary Linguistics. Chicago: The University of Chicago Press.
16. Preller, Anne. 2007a. Linear Processing with Pregroup Grammars. *Studia Logica* 87(2:3): 171–197.
17. —. 2007b. Toward Discourse Representation Via Pregroup Grammars. *JoLLI* 16: 173–194.

18. Preller, Anne and Joachim Lambek. 2007. Free compact 2-categories. *Mathematical Structures for Computer Sciences* 17(1): 1–32.
19. Pullum, Geoffrey and Gerald Gazdar. 1987. *The Formal Complexity of Natural Language*, chap. Natural Languages and Context Free Languages, 138–182. Studies in Linguistics and Philisophy. Dordrecht: Reidel Publishing Company.
20. Salvitch, Walter, Emmon Bach, William Marsh, and Gila Safran-Naveh. 1987. *The Formal Complexity of Natural Language*, chap. Introduction to Part III, 283–285. Studies in Linguistics and Philisophy. Dordrecht: Reidel Publishing Company.
21. Shieber, Stuart M. 1987. *The Formal Complexity of Natural Language*, chap. Evidence Against the Context-Freeness of Natural Language, 320–334. Studies in Linguistics and Philisophy. Dordrecht: Reidel Publishing Company.
22. Stabler, Edward. 2004. Varieties of crossing dependencies: Structure dependence and mild context sensitivity. *Cognitive Science* 28(5): 669–720.
23. —. 2008. Tupled Pregroup Grammars. In *Computational and Algebraic Approaches to Natural Language*, edited by Claudia Casadio and Joachim Lambek. Milano, Italia: Polimetrica.
24. Steedman, Mark. 1996. *Surface Structure and Interpretation, Linguistic Inquiry Monograph*, vol. 30. Cambridge, Massachusetts: MIT Press.