

Linear Processing with Pregroups

Anne Preller

LIRMM-CNRS

161, rue Ada

Montpellier, France

preller@lirmm.fr *

Abstract

Pregroup grammars have a cubic recognition algorithm. Here, we define a correct and complete recognition and parsing algorithm and give sufficient conditions for the algorithm to run in linear time. These conditions are satisfied by a large class of pregroup grammars, including grammars that handle coordinate structures and distant constituents.

1 Introduction

Pregroup grammars were introduced in [Lambek 99] as an algebraic tool for the syntactical analysis of sentences and shown to be weakly equivalent to context free grammars by [Buszkowski]. Therefore the usual cubic recognition algorithms of [CYK] or [Earley] for context-free grammars can be used for pregroup grammars after transforming them into rewrite-grammars. Cubic algorithms tailored to pregroups were given in 2004 independently by [Oehrle] and [Degeilh-Preller]. The algorithm proposed in the latter improves on the usual algorithms for context-free languages by the fact the constant factor of n^3 no longer depends on the number of rules or the number of symbols in the grammar and that it also works for infinite dictionaries. In particular it shows that proof search in compact bilinear logic is proportional to the cube of the length of the string of simple types.

However, cubic complexity easily exceeds human memory capacity and therefore natural languages are most likely to have grammars which permit linear processing. Two observations are important as far as pregroup grammars are concerned: A low number of types per word or a low number of basic types does not diminish the complexity of the recognition algorithm nor the number of different parsings. As an example, consider the rigid dictionary listing three words, each with a single type, say sa^ℓ , aa^ℓ and aa^ra respectively. The general algorithm still requires cubic time. Moreover, the number of different parsings

*The original publication is available at www.springerlink.com

increases exponentially with the length of the string. It seems appropriate to divide the task of finding a linear recognition and parsing algorithm as follows

1. choose a lexical entry for each word, i.e. choose a meaning
2. decide if the choice of types has a reduction to the sentence type and, if there is one, provide at least one such reduction.

We impose conditions on the dictionary concerning both aspects. After the basic notions on pregroups and reductions we present a dictionary covering distant dependencies and coordination in Section 3. as an example of the language fragments for which complete linear parsing is possible. In Section 4. we introduce a parsing algorithm and indicate a class of dictionaries for which it is complete. Restricting the class of dictionaries even more we make the parsing algorithm linear. These conditions are relaxed in Sections 5. where we define the minimal reduction, representative for all reductions to the sentence type. Finally in Section (6), we modify the earlier parsing algorithm by incorporating minimal reductions and define a class of dictionaries for which it is complete.

2 Geometry of derivations

We briefly recall the definition of pregroups and the construction of a freely generated pregroup defined in [Lambek 99]. As we are interested in parsing and semantical interpretation, we look at the actual derivations in a free pregroup concentrating on derivations that consist of generalized contractions only. The geometrical structure of these derivations is at the base of the semantical interpretation in Section 3.

A preordered monoid $\langle P, 1, \cdot, \rightarrow \rangle$ is a set with at least one element $1 \in P$, a binary operation \cdot and a binary relation \rightarrow satisfying for all $x, y, z, u, v \in P$

$$\begin{aligned} 1 \cdot x &= x = x \cdot 1 \\ (x \cdot y) \cdot z &= x \cdot (y \cdot z) \\ x &\rightarrow x \\ x \rightarrow y \text{ and } y \rightarrow z &\text{ imply } x \rightarrow z \\ x \rightarrow y &\text{ implies } u \cdot x \cdot v \rightarrow u \cdot y \cdot v. \end{aligned}$$

The dot denotes multiplication and is generally omitted. The arrow \rightarrow denotes the preorder.

A *pregroup* is a preordered monoid in which every element x has both a *left adjoint* x^ℓ and a *right adjoint* x^r satisfying

$$\begin{aligned} \text{(Contraction)} \quad & x^\ell x \rightarrow 1, \quad x x^r \rightarrow 1 \\ \text{(Expansion)} \quad & 1 \rightarrow x^r x, \quad 1 \rightarrow x x^\ell. \end{aligned}$$

One derives

- (1.) $x \rightarrow y$ if and only if $y^\ell \rightarrow x^\ell$ if and only if $y^r \rightarrow x^r$,
- (2.) $x \rightarrow y$ if and only if $x y^r \rightarrow 1$ if and only if $y^\ell x \rightarrow 1$.

The *free* pregroup $P(B)$ generated by a partially ordered set of *basic types* $B = \{a, b, \dots\}$ is characterized in [Lambek 99] as the preordered free monoid

generated from the set of *simple types* Σ consisting of the basic types and their *iterated adjoints*

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

where $a^{(0)}$ is identified with a . Note that the empty string is not a simple type. Elements $X \in P(B)$ are called *types*. They are strings of the form

$$X = a_1^{(z_1)} \dots a_k^{(z_k)},$$

where a_1, \dots, a_k are basic types and z_1, \dots, z_k are integers. The unit 1 denotes the empty string and multiplication is the same as concatenation. The left and right adjoints of a type are defined by

$$\begin{aligned} (a_1^{(z_1)} \dots a_k^{(z_k)})_\ell &= a_k^{(z_k-1)} \dots a_1^{(z_1-1)} \\ (a_1^{(z_1)} \dots a_k^{(z_k)})_r &= a_k^{(z_k+1)} \dots a_1^{(z_1+1)}. \end{aligned}$$

Hence, identifying $a \in B$ with $a^{(0)} \in \Sigma$ we have

$$a^{\ell\ell} = a^{(-2)}, a^\ell = a^{(-1)}, a = a^{(0)}, a^r = a^{(1)}, a^{rr} = a^{(2)} \text{ etc.}$$

If $t = a^{(z)}$ we call z the *iterator* of t .

Finally, the preorder on types is defined as the transitive closure of the union of the following three relations

$$\begin{aligned} \text{(Induced step)} \quad & Xa^{(z)}Y \rightarrow Xb^{(z)}Y \\ \text{(Generalized contraction)} \quad & Xa^{(z)}b^{(z+1)}Y \rightarrow XY \\ \text{(Generalized expansion)} \quad & XY \rightarrow Xa^{(z+1)}b^{(z)}Y \end{aligned}$$

where either z is even and $a \rightarrow b$ or z is odd and $b \rightarrow a$. Here X, Y are (possibly empty) strings of simple types and a and b are basic.

As usual, a *substring* of a string $s_1 \dots s_n$ has the form $s_{i_0} \dots s_{i_k}$ where $1 \leq i_0 < \dots < i_k \leq n$. A *segment* $s_l \dots s_m$ is a substring such that $l = i_0$, $l+1 = i_1$, \dots , $l+k = i_k$ and $m = l+k$. By convention, if $m < l$, then $s_l \dots s_m$ stands for the empty segment.

The property which makes the theory of pregroups decidable is expressed in the so-called

Switching Lemma (Proposition 2 of [Lambek 99]):

Let s_1, \dots, s_n and t_1, \dots, t_m be simple types. Then $s_1 \dots s_n \rightarrow t_1 \dots t_m$ if and only if there are a substring $s_{i_1} \dots s_{i_k}$ of $s_1 \dots s_n$ and a substring $t_{i_1} \dots t_{i_k}$ of $t_1 \dots t_m$ such that

$$\begin{aligned} s_1 \dots s_n &\rightarrow s_{i_1} \dots s_{i_k} \rightarrow t_{i_1} \dots t_{i_k} \rightarrow t_1 \dots t_m, \\ s_{i_p} &\rightarrow t_{i_p}, \text{ for } 1 \leq p \leq k, \end{aligned}$$

where $s_{i_1} \dots s_{i_k}$ is obtained from $s_1 \dots s_n$ by generalized contractions only, $t_1 \dots t_m$ is obtained from $t_{i_1} \dots t_{i_k}$ by generalized expansions only and $t_{i_1} \dots t_{i_k}$ is obtained from $s_{i_1} \dots s_{i_k}$ by induced steps only.

A string of simple types $s_1 \dots s_n$ is *irreducible* if no two adjacent simple types satisfy $s_i s_{i+1} \rightarrow 1$. In particular, the empty string 1 is irreducible. A type $s_{i_1} \dots s_{i_k}$ is an *irreducible form* of $s_1 \dots s_n$ if it is irreducible and obtained from $s_1 \dots s_n$ by generalized contractions only. If s and the s_i 's are simple for $1 \leq i \leq n$, then $s_1 \dots s_n \rightarrow s$ if and only if for some i , s_i is an irreducible form of $s_1 \dots s_n$ and $s_i \rightarrow s$.

In linguistic applications, a dictionary lists each word with types belonging to a free pregroup. Then a string of words is a sentence, if one can assign to every word a type from the dictionary such that the concatenation $t_1 \dots t_m$ of these types has a derivation to a distinguished basic type \mathbf{s} , called sentence type. In view of Property (2.) above, $t_1 \dots t_m \rightarrow \mathbf{s}$ is equivalent to $t_1 \dots t_m \mathbf{s}^r \rightarrow 1$. By the Switching Lemma, a derivation to the empty string can be obtained by generalized contractions only. For example, consider the set of basic types

$$B = \{\nu_{\mathbf{s}}, \pi_{3\mathbf{s}}, o, n_{\mathbf{s}}, c_{\mathbf{s}}, \mathbf{s}_1, \mathbf{s}\},$$

where $\nu_{\mathbf{s}} \rightarrow \pi_{3\mathbf{s}}, \nu_{\mathbf{s}} \rightarrow o, n_{\mathbf{s}} \rightarrow \pi_{3\mathbf{s}}, n_{\mathbf{s}} \rightarrow o, \mathbf{s}_1 \rightarrow \mathbf{s}$. Here ν_3 is the type for proper names singular, $\pi_{3\mathbf{s}}$ for 3-person singular subjects, o for direct object complements of a verb, $n_{\mathbf{s}}$ for singular noun phrases, $c_{\mathbf{s}}$ for singular common nouns and finally, \mathbf{s} and \mathbf{s}_1 are sentence types, the latter for statements in the present tense, the former for statements when the tense does not matter. Then the dictionary may list

$$\begin{aligned} \text{Mary} &: \nu_{\mathbf{s}} \\ \text{buys} &: \pi_{3\mathbf{s}}^r \mathbf{s}_1 o^\ell \\ a &: n_{\mathbf{s}} c_{\mathbf{s}}^\ell \\ \text{book} &: c_{\mathbf{s}} \end{aligned}$$

Therefore, the grammar generates the sentence

$$\text{Mary} \quad \text{buys} \quad a \quad \text{book} \\ (\nu_{\mathbf{s}}) \quad (\pi_{3\mathbf{s}}^r \quad \underline{\mathbf{s}_1 o^\ell}) \quad (\underline{n_{\mathbf{s}} \quad c_{\mathbf{s}}^\ell}) \quad (c_{\mathbf{s}}) \quad \mathbf{s}^r$$

The underlinks indicate the generalized contractions used in the derivation, namely $\nu_{\mathbf{s}} \pi_{3\mathbf{s}}^r \rightarrow 1$, $o^\ell n_{\mathbf{s}} \rightarrow 1$, $c_{\mathbf{s}}^\ell c_{\mathbf{s}} \rightarrow 1$ and $\mathbf{s}_1 \mathbf{s}^r \rightarrow 1$. In fact, the underlinks together uniquely determine a derivation to the empty string or more generally, a derivation of $s_1 \dots s_n$ to a substring $s_{i_1} \dots s_{i_p}$ consisting of generalized contractions only. We decompose such a derivation into a geometrical part, consisting of the set of underlinks, and an algebraic part, consisting of the generalized contractions. The *geometrical part* R , also called *reduction*, has the following definition

Definition 1. A *reduction* R with respect to n is a set of two-element subsets $\{i, k\} \subseteq \{1, \dots, n\}$, called *underlinks*, which satisfy

- 1) for every i with $1 \leq i \leq n$, there is at most one k such that $\{i, k\} \in R$,
- 2) if $\{i, k\} \in R$ and $i < l < k$ then there is $i < m < k$ such that $\{l, m\} \in R$.

The *algebraic part* consists of the generalized contractions

- 3) $s_i s_k \rightarrow 1$, for $i < k$ such that $\{i, k\} \in R$

Let $\{i_1, \dots, i_p\}$ be the subset of $\{1, \dots, n\}$ consisting of the elements i_l for which there is no k such that $\{i_l, k\}$ belongs to R . A reduction R is called a *transition* from $s_1 \dots s_n$ to $s_{i_1} \dots s_{i_p}$, written

$$R : s_1 \dots s_n \Rightarrow s_{i_1} \dots s_{i_p}$$

if all three conditions above hold. If the substring $s_{i_1} \dots s_{i_p}$ cannot be contracted any further, it is called an *irreducible form* of $s_1 \dots s_n$.

The empty string 1 and every simple type is irreducible. A string of simple types has at least one irreducible form. Some strings have several irreducible forms, and there may be different reductions bringing it to the same irreducible form, e.g. the first two reductions below are transitions to the empty string, the third reduction is a transition to the irreducible $a^r a$

$$1) \quad \underbrace{a^\ell a \underline{a^\ell a} a^r}_a \quad 2) \quad \underline{a^\ell a} \underline{a^\ell a a^r}_a \quad 3) \quad \underline{a^\ell a a^\ell a} a^r a \quad (I)$$

On the other hand, the same reduction may constitute a transition to the empty string for quite different strings of simple types. For example, assume $a \rightarrow b$ and consider the three different transitions below

$$\underbrace{b^\ell a \underline{b^\ell a} a^r}_a, \quad \underline{b^\ell a} \underline{b^\ell a a^r}_a, \quad \underline{c \underline{c^\ell c^r c^{rr}}}_c c^r.$$

The first and the third transition have the same links, the common geometrical part is $R = \{\{1, 6\}, \{2, 5\}, \{3, 4\}\}$. The second transition, however, has a different reduction $R' = \{\{1, 2\}, \{3, 6\}, \{4, 5\}\}$. This geometrical difference corresponds to different meanings of the same sentence, as illustrated by the examples (1a) and (1b) of the next section.

Using the graphical representation of $\{i, k\} \in R, i < k$, by an underlink

$$\dots \underline{s_i \dots s_k} \dots,$$

we may describe a reduction as a planar graph which has a linearly ordered set of vertices labeled by simple types such that

- there are no loops
- every $i \notin \{i_1, \dots, i_p\}$ is endpoint of exactly one underlink
- underlinks do not cross
- there is no underlink $\{i, k\}$ such that $i \leq i_l \leq k$ for $1 \leq l \leq p$.

If $i < k$, we call i or s_i the *left endpoint* and k or s_k the *right endpoint* of the link $\{i, k\}$. If the reduction R is a transition $R : s_1 \dots s_n \Rightarrow s_{i_1} \dots s_{i_p}$, then the iterator of the right endpoint of an underlink is the successor of the iterator of its left endpoint: Indeed, if $\{i, k\} \in R$ and $i < k$, then the algebraic condition $s_i s_k \rightarrow 1$ implies that $s_i = a^{(z)}$ and $s_k = b^{(z+1)}$ for some integer z and basic types a and b .

Finally, we remark that an arbitrary transition can always be obtained as the union of transitions to the empty string. Indeed, let $R_l, 0 \leq l \leq p$, be

reductions such that $R_0 : s_1 \dots s_{i_1-1} \Rightarrow 1$, $R_l : s_{i_l+1} \dots s_{i_{l+1}-1} \Rightarrow 1$ for $1 \leq l < p$, and $R_p : s_{i_p+1} \dots s_n \Rightarrow 1$. Then the union

$$R = \bigcup \{R_l : 0 \leq l \leq p\}$$

is a reduction such that $R : s_1 \dots s_n \Rightarrow s_{i_1} \dots s_{i_p}$.

3 Coordinate Structures and Unbounded Dependencies

We illustrate parsing with pregoup grammars by a few examples on coordination and unbounded dependencies involving the relative pronoun. The semantical dependencies are expressed by (multi)links. Consider the dictionary

$$\begin{aligned} \text{Mary} & : \nu_s \\ \text{bought} & : \pi^r s_2 o^\ell, \pi^r \hat{o}^r \hat{s}_2, \hat{\pi}^r \hat{s}_2 o^\ell \\ \text{a} & : n_s c_s^\ell \\ \text{horse} & : c_s \\ \text{which} & : c_s^r c_s \hat{s}^\ell \hat{o}, c_s^r c_s \hat{s}^\ell \hat{\pi}_{3s} \\ \text{John} & : \nu_s \\ \text{detests} & : \pi_{3s}^r s_1 o^\ell, \pi_{3s}^r \hat{o}^r \hat{s}_1, \hat{\pi}_{3s}^r \hat{s}_1 o^\ell \end{aligned}$$

Recall that o is the type of a direct object complement, π_{3s} stands for the subject third person singular and π for the subject if the person and number do not matter. The dictionary lists three types for the definite verb form. This reflects the grammatical distinction between statements and relative clauses. In particular, the basic type \hat{s}_1 is the sentence type for relative clauses in the present and \hat{s} the type for relative clauses if the tense does not matter. The other new basic types are used as dummies, namely \hat{o} for a direct object complement, $\hat{\pi}_{3s}$ for a third person subject in the singular, $\hat{\pi}$ if the number and person do not matter. The set of basic types occurring in the dictionary is

$$B = \{s, s_1, s_2, \pi_{3s}, \pi, o, \nu_s, n_s, c_s, \hat{s}_1, \hat{s}, \hat{\pi}_{3s}, \hat{\pi}, \hat{o}\}$$

where

$$\nu_s \rightarrow \pi_{3s}, n_s \rightarrow \pi_{3s}, \nu_s \rightarrow o, n_s \rightarrow o, \pi_{3s} \rightarrow \pi, s_1 \rightarrow s, s_2 \rightarrow s, \hat{s}_1 \rightarrow \hat{s}.$$

Then we use different type assignments for the relative pronoun *which*, according to its role as an object complement or subject in the relative clause.

Example I

$$\text{Mary bought a horse which John detests} \\ (\nu_s) (\pi^r s_2 o^\ell) (n_s c_s^\ell) (c_s) (c_s^r c_s \hat{s}^\ell \hat{o}) (\nu_s) (\pi_{3s}^r \hat{o}^r \hat{s}_1) s^r \quad (1a)$$

$$\text{Mary bought a horse which detests John} \\ (\nu_s) (\pi^r s_2 o^\ell) (n_s c_s^\ell) (c_s) (c_s^r c_s \hat{s}^\ell \hat{\pi}_{3s}) (\hat{\pi}_{3s}^r \hat{s}_1 o^\ell) (\nu_s) s^r \quad (1b)$$

The two reductions above differ only by the links whose left endpoint is under *which* or later. This becomes more evident with the following abbreviation, called a *multilink*:

$$\underline{YX^r} \text{ for } \underbrace{t_1 \dots t_p s_p^r \dots s_1^r}_{\text{multilink}}, \text{ where } X = s_1 \dots s_p, Y = t_1 \dots t_p.$$

Replacing the two links under $\hat{s}^\ell \hat{o} \dots \hat{o}^r \hat{s}_1$ by a multilink and omitting the types in the first example, we get the dependencies

$$\begin{array}{c} \text{which John detests} \\ \dots \boxed{\quad \quad \quad} \end{array} \quad (1a)$$

giving the correct the semantical analysis of the relative clause. Applying the same simplifications to the second example, we obtain the dependencies

$$\begin{array}{c} \text{which detests John} \\ \dots \boxed{\quad \quad \quad} \end{array} \quad (1b)$$

Dummies are entities not explicitly named in a sentence but are implied by the statement. Their syntactical role is to fill what [Gazdar] calls a hole. It is pointed out in [*loc. cit.*] that holes have a syntactic category. In pregroup grammars, the hole is marked by a right or left adjoint of a dummy type and therefore also has a syntactic representation. We take advantage of this fact when analysing coordinate structures conjoined by *and*. The sample sentences are adapted from [*loc. cit.*].

$$\text{and} : x^r x x^\ell, x = \hat{o}^r \hat{s}_1 \quad (2a)$$

$$\text{and} : x^r x x^\ell, x = \hat{\pi}_{3s}^r \hat{s}_1 \quad (2b)$$

Consider the case where the hole is the object complement of the verb in the relative clause, i.e.

$$x = \hat{o}^r \hat{s}_1.$$

Let $y = \hat{o}^r \hat{s}$. Then

$$\text{detests, loves} : \pi_{3s}^r x$$

and

$$\text{which} : c_s^r c_s y^\ell.$$

It follows that $x \rightarrow y$ and we find the following reduction

$$\text{Mary bought a horse which John detests and Jo loves} \\ (\nu_s) (\pi_{3s}^r s_2 o^\ell) (n_s c_s^\ell) (c_s) (c_s^r c_s y^\ell) (\nu_s) (\pi_{3s}^r x) (x^r x x^\ell) (\nu_s) (\pi_{3s}^r x) s^r.$$

The conjunction *and* distributes the holes both times to the position of the second argument of the relation expressed by the verb. This position corresponds by convention to the direct object of the verb:

$$\text{horse which John detests and Jo loves} \\ (\quad) (\quad) (\quad) (\quad) (\quad) (\quad) (\quad) (\quad) (\quad) (\quad)$$

Next, consider the case where the hole is the subject of the relative clause:

$$x = \hat{\pi}_{3s}^r \hat{s}_1, y = \hat{\pi}_{3s}^r \hat{s}.$$

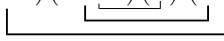
Then

$$\text{detests, loves} : x o^\ell \text{ and } \text{which} : c_s^r c_s y^\ell$$

As before, we find $x \rightarrow y$ and with this construct the reduction of the sentence

$$\text{Mary bought a horse which detests John and loves Jo} \\ (\nu_s) (\pi_{3s}^r s_2 o^\ell) (n_s c_s^\ell) (c_s) (c_s^r c_s y^\ell) (x o^\ell) (\nu_s) (x^r x x^\ell) (x o^\ell) (\nu_s) s^r$$

This time, the holes are distributed by the conjunction *and* to the first argument place of the verb, i.e. to the position of the subject:

which detests John and loves Jo
 $(\quad)(\quad)(\quad)(\quad)(\quad)$


However the two different kinds of holes cannot be conjoined:

*Mary bought a horse which John detests and loves Jo

For a proof, it suffices to consider every type assignment from our dictionary and verify that it has no reduction to the sentence type. Therefore *which* cannot be both subject and object of verb phrases constituting the relative clause.

4 Lazy Parsing

We give a ‘lazy’ recognition and parsing algorithm and define sufficient conditions on the dictionaries for which it is linear and complete.

In the following, the partially ordered set B and the free pregroup $P(B)$ generated by B are fixed. As usual, a *dictionary* \mathcal{D} over B for a set of words V is a map from V to the set of subsets of $P(B)$. Instead of $T_l \in \mathcal{D}(v_l)$ we may write $v_l : T_l$ and call it a *lexical entry*. We distinguish a basic type s , the so-called *sentence type*. A string of types $T_1 \dots T_n$ is called a *type assignment* for $v_1 \dots v_n$ if $v_l : T_l$ is a lexical entry for $1 \leq l \leq n$. A *parsing* of a string $v_1 \dots v_n$ consists of a type assignment $T_l \in \mathcal{D}(v_l)$ and a reduction of $T_1 \dots T_n$ to s .

We begin by describing an algorithm which combines the search for a reduction with type assignment. It processes by stages, reading the string of words $v_1 \dots v_n$ from left to right. At each stage, it either chooses a type for the word under examination or processes the assigned type by reading its simple types from left to right. The result is a reduction of the type processed so far to an irreducible type.

The set of *stages* associated to $v_1 \dots v_n$ consists of an initial stage s_{in} and of triples $s = (l, T_1 \dots T_l, p)$ where

l is the number of the word v_l being processed

$T_k = a_{k1} \dots a_{kq_k}$ in $\mathcal{D}(v_k)$, $1 \leq k \leq l$, a type assignment for $v_1 \dots v_l$

p a *position*, $0 \leq p \leq q_l$.

The stages are partially ordered as follows

$s_{in} < s$ for all s

$(l, T_1 \dots T_l, p) \leq (l', T'_1 \dots T'_{l'}, p') \Leftrightarrow l \leq l', T_k = T'_k$ for $1 \leq k \leq l, p \leq p'$.

We remark that all non-initial stages s have a unique immediate predecessor, which we denote by $s - 1$, i.e.

$$(l, T_1 \dots T_l, p) - 1 = \begin{cases} (l, T_1 \dots T_l, p - 1), & \text{if } 1 \leq p; \\ (l - 1, T_1 \dots T_{l-1}, q_{l-1}) & \text{if } p = 0 \text{ and } l > 1, \\ s_{in}, & \text{if } p = 0 \text{ and } l = 1. \end{cases}$$

It follows that the set of stages smaller than or equal to a given stage s is totally ordered.

This total order can be used to control the way how the algorithm moves through the stages and define the actual position $p(s)$ and the type $a_{p(s)}$ read

at this position. At the initial stage $p(s_{in}) = 0$, $a_0 = 1$. A stage of the form $(l, T_1 \dots T_l, 0)$, $1 \leq l \leq n$, is called a *downloading stage* and serves to choose a type $T_l \in \mathcal{D}(v_l)$ as soon as the word v_l has been given. At a downloading stage $s = (l, T_1 \dots T_l, 0)$, the examined position remains unchanged

$$p(s) = p(s-1) = q_1 + \dots + q_{l-1} + 0.$$

After downloading, the string of simple types T_l is read from left to right. Each stage which is not initial and not downloading is called a *testing stage*. To reach the testing stage $s = (l, T_1 \dots T_l, p)$, $p \geq 1$, the preceding position $p(s-1)$ is incremented by 1:

$$p(s) = p(s-1) + 1 = q_1 + \dots + q_{l-1} + p.$$

It follows that the simple type occupying this position satisfies

$$a_{p(s)} = a_{lp}.$$

The type processed at stage $s = (l, T_1 \dots T_l, p)$ can be defined inductively by

$$\begin{aligned} T(s_{in}) &= 1 && = a_0 \\ T(s) &= T(s-1) && = T_1 \dots T_{l-1} && \text{if } 0 = p \\ T(s) &= T(s-1)a_{p(s)} && = T_1 \dots T_{l-1}a_{l1} \dots a_{lp} && \text{if } 0 < p \end{aligned}$$

More generally, for every i such that $1 \leq i \leq p(s)$ there are a unique k and a unique p' such that $1 \leq k \leq l$, $1 \leq p' \leq q_k$, $i = q_1 + \dots + q_{k-1} + p'$ and

$$a_i = a_{kp'}.$$

The simple type $a_{p(s)}$ is tested for generalized contraction with the last not contracted type in the string. This test can be done in one time unit by accessing the partial order relation on the set of basic types. If it fails, $p(s)$ is added on the top of the stack indicating that $a_{p(s)}$ is the latest not (yet) contracted type. The other data remain unchanged. If the test succeeds, the stack is popped and the link consisting of the contracting positions is added to the reduction computed so far.

A stack $S(s)$ constructed by this algorithm at stage s is an ordered pair $\langle S', i \rangle$ where i is a non-negative integer and S' is either the empty stack \emptyset or the stack of a preceding stage. As the test is only performed for non-initial and non-downloading stages, all positions i stored in the stack at stage s correspond to a unique stage $s' \leq s$ for which $a_{p(s')} = i$. The functions *top* and *pop* send a stack $\langle S', i \rangle$ to its top i and its tail S' respectively. They are undefined for the empty stack. A stack of positions defines a substring of $a_1 \dots a_n$ as follows

$$\begin{aligned} \overline{\langle \emptyset, 0 \rangle} &= 1 \\ \overline{\langle S', i \rangle} &= \overline{S'}a_i \end{aligned}$$

Definition 2. Lazy Parsing Algorithm

▼ At the initial stage, let

$$S(s_{in}) = \langle \emptyset, 0 \rangle, R(s_{in}) = \emptyset$$

▼ At a downloading stage $s = (l, T_1 \dots T_l, 0)$, the stack and reduction remain unchanged

$$S(s) = S(s-1), \quad R(s) = R(s-1)$$

▼ If $s = (l, T_1 \dots T_l, p)$ is not downloading and not initial, let $t(s-1) = \text{top}(S(s-1))$. Then

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

$$R(s) = \begin{cases} R(s-1) \cup \{\{t(s-1), p(s)\}\}, & \text{if } a_{t(s-1)}a_{p(s)} \rightarrow 1 \\ R(s-1), & \text{else} \end{cases}$$

The stack can increase and decrease during processing, but it will never get empty. Indeed, if $S(s-1) = \langle \emptyset, 0 \rangle$ then the test will fail, as $a_0a_{p(s)} = a_{p(s)} \not\rightarrow 1$. Hence $S(s) = \langle \langle \emptyset, 0 \rangle, p(s) \rangle$. Note also that the entries of the stack form a strictly increasing string of integers between 0 and $p(s)$. Moreover, the set R computed at stage s is a reduction of the string processed so far to an irreducible string, which is defined by the stack S :

Theorem 3 (Irreducible Form). *For every stage $s = (l, T_1 \dots T_l, p)$, the string $\overline{S(s)}$ associated to the stack $\overline{S(s)}$ is an irreducible substring of $T(s)$ and $R(s)$ is a transition from $T(s)$ to $\overline{S(s)}$.*

PROOF: Use induction on the linearly ordered set of the predecessors of s . Clearly, the property holds for the initial stage. Now assume that the property holds up to $s-1$. If s is a downloading stage, the property follows immediately from the induction hypothesis. If s is a testing stage, let $t(s) = \text{top}(S(s))$ and decompose the processed string $T(s) = a_1 \dots a_{p(s)}$ into an initial and a terminal segment

$$a_1 \dots a_{p(s)} = (a_1 \dots a_{t(s)})(a_{t(s)+1} \dots a_{p(s)})$$

and show that $R(s)$ is the union of two reductions

- (1) $R'(s) : a_1 \dots a_{t(s)} \Rightarrow \overline{S(s)}$
- (2) $R''(s) : a_{t(s)+1} \dots a_{p(s)} \Rightarrow 1$.

For that purpose, distinguish two cases:

Case I : $a_{t(s-1)}a_{p(s)} \not\rightarrow 1$.

Then $t(s) = p(s)$, $S(s) = \langle S(s-1), p(s) \rangle$ and $R(s) = R(s-1)$ and $\overline{S(s)} = \overline{S(s-1)}a_{t(s)}$. Then $a_{t(s)+1} \dots a_{p(s)}$ is the empty string and the restriction $R''(s)$ of $R(s)$ to this string is the empty set. Therefore (2) holds trivially.

Next, from $a_{t(s-1)}a_{t(s)} \not\rightarrow 1$ and the induction hypothesis follows that $\overline{S(s)} = \overline{S(s-1)}a_{t(s)}$ is an irreducible form of $a_1 \dots a_{p(s)} = (a_1 \dots a_{p(s)-1})a_{p(s)}$. Recall that $p(s-1) = p(s) - 1$ and that $R(s-1) : a_1 \dots a_{p(s-1)} \Rightarrow \overline{S(s-1)}$ by induction hypothesis. Hence $R(s) : a_1 \dots a_{p(s)} \Rightarrow \overline{S(s-1)}a_{p(s)} = \overline{S(s)}$. As $R'(s) = R(s)$, (1) holds.

Case (II) : $a_{t(s-1)}a_{p(s)} \rightarrow 1$.

In this case,

$$S(s-1) = \langle S', t(s-1) \rangle$$

where S' is the stack computed at some stage s' preceding $s-1$. Moreover,

$$S(s) = \text{pop}(S(s-1)) = S'$$

and

$$R(s) = R(s-1) \cup \{t(s-1), p(s)\}.$$

Then $t(s) = \text{top}(S')$, $\overline{S(s)} = \overline{S'}$ and $\overline{S(s-1)} = \overline{S'} a_{t(s-1)} = \overline{S(s)} a_{t(s-1)}$. By induction hypothesis

- (1) $R'(s-1) : a_1 \dots a_{t(s-1)} \Rightarrow \overline{S(s-1)}$
- (2) $R''(s-1) : a_{t(s-1)+1} \dots a_{p(s-1)} \Rightarrow 1$.

Now decompose $a_1 \dots a_{p(s)}$ as follows

$$a_1 \dots a_{p(s)} = (a_1 \dots a_{t(s)}) (a_{t(s)+1} \dots a_{t(s-1)-1}) a_{t(s-1)} (a_{t(s-1)+1} \dots a_{p(s-1)}) a_{p(s)}.$$

Note that $R'(s)$, the restriction of $R(s)$ to $\{1, \dots, t(s)\}$, is also the restriction of $R(s-1)$ to $\{1, \dots, t(s)\}$. Hence

$$R'(s) : a_1 \dots a_{t(s)} \Rightarrow \overline{S(s)}.$$

Next, let $R^*(s)$ be the restriction of $R(s-1)$ to $\{t(s)+1, \dots, t(s-1)-1\}$. It reduces the segment $a_{t(s)+1} \dots a_{t(s-1)-1}$ between two consecutive positions of $\overline{S(s-1)}$ to 1. Finally, the restriction of $R(s)$ to $\{t(s)+1, \dots, p(s)\}$ is

$$R''(s) = R^*(s) \cup R''(s-1) \cup \{t(s-1), p(s)\}.$$

Hence

$$R''(s) : (a_{t(s)+1} \dots a_{t(s-1)-1}) (a_{t(s-1)}) (a_{t(s-1)+1} \dots a_{p(s-1)}) a_{p(s)} \Rightarrow 1.$$

We remark that the number of steps necessary to compute $S(s)$ from its predecessor $S(s-1)$ is bounded by a constant which depends only of the dictionary. Hence we have the following property

Corollary 4. *For every stage $s = (n, T_1 \dots T_n, q_n)$, Lazy Parsing computes the irreducible form $\overline{S(s)}$ and the reduction $R(s) : T_1 \dots T_n \rightarrow \overline{S(s)}$ in time proportional to the length of $T_1 \dots T_n$.*

Note that Lazy Parsing also works on infinite dictionaries. Consider the identity map on the set of simple types Σ , i.e. the dictionary with the entries $t : t$, for $t \in \Sigma$. The set of stages associated to the string $t_1 \dots t_n$ is $\{1, \dots, n\}$ and Lazy Parsing produces a derivation of $t_1 \dots t_n$ to one of its irreducible forms.

Backward Lazy Parsing, that is to say reading a string of words from right to left, is defined similarly. In particular, the positions stored in the stack are decreasing and the test for contraction is $a_{p(s)} a_{t(s-1)} \rightarrow 1$. The appropriate variant of the preceding theorem then holds for Backward Lazy Parsing. In general, the computed irreducible form may be different from that computed by ‘forward’ lazy parsing. For example, backward parsing finds the second reduction of (I) in Section (2) and forward parsing the third.

If the computed irreducible form happens to be the sentence type, the algorithm gives a parsing of the sentence. If this is not the case, we cannot conclude in general that $T_1 \dots T_n$ has no reductions to the sentence type. Hence, the algorithm is not complete unless we impose some conditions on the dictionary. The rest of this section is devoted to define a class of pregroup grammars for which the Parsing Algorithm is complete. An even more comprehensive class of pregroup grammars can also be parsed in linear time by a slight variant of our algorithm, as we shall see in the next section.

Definition 5. (Complexity:)

A connected component \mathcal{C} of basic types has *complexity* $K \geq 0$, if there is an integer u satisfying the following two conditions

- whenever $b \in \mathcal{C}$ and $b^{(z)}$ occurs in \mathcal{D} , then $u \leq z \leq u + K$
- there are $b_i \in \mathcal{C}$ for which $b_i^{(u+i)}$ occurs in \mathcal{D} , for $0 \leq i \leq K$.

A *simple type* $t = a^{(z)}$ has complexity K if a belongs to a connected component of complexity less than or equal to K . A string of simple types has complexity K if every simple type in the string has complexity K . A *grammar is of complexity* K if every type in its dictionary has complexity K .

Our sample dictionary of Section (3) has complexity 2. Indeed, the simple types occurring in the dictionary are either basic or right or left adjoints of basic types. Without the words *which* and *and*, it would be of complexity 1.

Grammars of complexity 1 are interesting because of their algorithmic properties, but they are not expressive enough. The next class, that of complexity 2, is as expressive as the whole class of pregroup grammars. Indeed, every pregroup grammar is strongly equivalent to a dictionary using only basic types, right adjoints of basic types and left adjoints of basic types, see [Preller 07]. Here strongly equivalent means not only that the grammars generate the same sentences but also that a reduction of a type-assignment in one grammar is also a reduction for some type assignment in the other grammar. Hence the links under the words are the same in both grammars.

We will show that the Lazy Parsing Algorithm is complete for dictionaries of complexity 1 and indeed for an even larger class, the *linear* dictionaries.

Definition 6. (Critical types, critical triples and linear types)

A simple type $t = c^{(z+1)}$ is *critical* if there are simple types a, b in the connected component of c such that $a^{(z-1)}$ and $b^{(z)}$ occur in \mathcal{D} .

A simple type $t = b^{(z)}$ occurring in \mathcal{D} is said to be *right-only* if $c^{(z+1)}$ does not occur in \mathcal{D} for all c in the connected component of b . The notion of *left-only* is defined similarly.

Let $T = t_1 \dots t_n$ be a string of simple types and assume that $1 \leq i < j < k \leq n$. We say that t_i, t_j, t_k is a *critical triple* if

$$\begin{aligned} t_i t_j &\rightarrow 1, \quad t_j t_k \rightarrow 1, \\ t_{i+1} \dots t_{j-1} &\rightarrow 1, \quad t_{j+1} \dots t_{k-1} \rightarrow 1 \quad . \end{aligned}$$

A string of simple types without critical triples is called *linear*. A dictionary is *linear*, if all type assignments with a reduction to the sentence type are linear.

Compare the notion of a critical triple with that of a critical type. The former is relative to a given string of simple types, whereas the latter depends only of the dictionary. The last of a critical triple is necessarily a critical type. If the dictionary has complexity 2, the last of a critical triple $\dots t_i \dots t_j \dots t_k \dots$ is right-only and the first left-only. A string is necessarily linear if no critical type occurs in it. Hence dictionaries of complexity 1 are linear.

Lemma 7 (Uniqueness of Links in Linear Strings). *Suppose $t_1 \dots t_m$ is a linear string of simple types. Then for every position k there is at most one i satisfying*

$$i < k, t_i t_k \rightarrow 1 \text{ and } t_{i+1} \dots t_{k-1} \rightarrow 1,$$

or

$$i > k, t_k t_i \rightarrow 1 \text{ and } t_{k+1} \dots t_{i-1} \rightarrow 1.$$

PROOF: Note that k can not be both the right endpoint of a link $\{i, k\}$ and the left endpoint of a link $\{k, j\}$, because if this was the case, the string would have the critical triple $t_i \dots t_k \dots t_j$. Show that k can not be endpoint of two different links using induction on the length m of the longer link, i.e. $m = \max\{|k - i|, |k - j|\}$. If $m = 1$, t_k would be linked to t_{k-1} and to t_{k+1} , which is impossible as we just saw. If $m = 2$ we would have a link $\{i, i + 2\}$ with $t_{i+1} \rightarrow 1$, which is impossible for a simple type. Suppose the property holds up to $m - 1$, with $3 \leq m$. Assume first that t_k is right endpoint of two different links, i.e. that for some $i, j \geq 1$

$$\begin{aligned} i &< j < k \\ t_i t_k &\rightarrow 1 \\ t_{i+1} \dots t_{k-1} &\rightarrow 1 \\ t_j t_k &\rightarrow 1 \\ t_{j+1} \dots t_{k-1} &\rightarrow 1 \end{aligned}$$

By hypothesis, there are reductions R and R^* such that $R : t_i \dots t_k \Rightarrow 1$ and $\{i, k\} \in R$ and $R^* : t_j \dots t_k \Rightarrow 1$ and $\{j, k\} \in R^*$. Using underbrackets for the links of R and underbraces for the links of R^* , the situation is represented by the graph

$$\dots t_i \dots \underbrace{t_j \dots t_k}_{\text{link of } R} \dots$$

Then R links j to some position k' with $i + 1 \leq k' \leq k - 1$. Note that k' is necessarily to the right of j , as otherwise the string would have the critical triple

$$\dots \underbrace{t_{k'} \dots t_j}_{\text{link of } R} \dots t_k \dots, \text{ which is impossible.}$$

Hence, $j < k' < k$ and

$$\dots t_i \dots \underbrace{t_j \dots t_{k'}}_{\text{link of } R} \dots t_k \dots$$

Then t_j would be left endpoint of two different links. As $|j - k| < |i - k|$, this contradicts the induction hypothesis. The case where t_k is left endpoint is similar.

This local property of unique links has useful global consequences. The first one is the following Lemma

Lemma 8 (Uniqueness of Reductions of Linear Strings). *Every linear string of simple types $t_1 \dots t_m$ has a unique irreducible form $t_{i_1} \dots t_{i_p}$ and a unique reduction R to that irreducible form. This reduction is computed by Lazy Parsing*

PROOF: If $t_1 \dots t_m$ is irreducible, the property holds trivially. Else, note that in all reductions to an irreducible form, the links with neighboring endpoints must be identical. Indeed, let R and R' be reductions such that $R : t_1 \dots t_m \Rightarrow t_{i_1} \dots t_{i_p}$ and $R' : t_1 \dots t_m \Rightarrow t_{j_1} \dots t_{j_q}$ with $t_{i_1} \dots t_{i_p}$ and $t_{j_1} \dots t_{j_q}$ irreducible. Assume $\{k-1, k\} \in R$. As $t_{k-1}t_k \rightarrow 1$, one of $k-1$ or k must belong to a link in R' . If it is k , it can not be the left endpoint of a link $\{k, i\} \in R'$, because otherwise the string would have a critical triple, contradicting linearity. Hence R' can link k only to some position $i < k$. By the preceding lemma, it follows that $i = k-1$. Similarly, if it is $k-1$ that is linked in R' to some position, the latter is necessarily the position k . Hence R and R' coincide on all links with neighboring endpoints. Omitting these links from R and R' and the corresponding segments $t_{k-1}t_k$ from $t_1 \dots t_m$, the new reductions are reductions of the new string to the same irreducible forms $t_{i_1} \dots t_{i_p}$ and $t_{j_1} \dots t_{j_q}$ respectively. It follows by induction hypothesis, that the same positions are linked by R and by R' . But then the positions that are not linked also coincide in both reductions. Therefore $p = q$, $i_k = j_k$, for $1 \leq k \leq p$ and $t_{i_1} \dots t_{i_p} = t_{j_1} \dots t_{j_q}$.

Theorem 9 (Linear Completeness). *A string of words from a linear dictionary $v_1 \dots v_n$ is a sentence if and only if at some final stage $s = (n, T_1 \dots T_n, q_n)$, the reduction $R(s)$ reduces $T_1 \dots T_n$ to the sentence type. Moreover, for a given final stage, the reduction has been computed by Lazy Parsing in time proportional to the length of the corresponding type assignment.*

PROOF: The last assertion follows from the fact that the definition of $R(s)$ only involves the predecessors of s and that for each stage the number of computation steps is constant. The rest is a straightforward consequence of Theorem 3 and Lemma 8.

Hence Lazy Parsing finds the unique reduction of a linear string to an irreducible form. The same algorithm does as well for certain strings with critical triples. Consider a string which has all critical types at the end, like

$$bb \underbrace{b^\ell b, b^\ell b, b^r b^r}_{\text{critical types}}.$$

Such a string has at most one reduction to the empty string, which is computed by Lazy Parsing.

Lemma 10 (Quasi-Linearity). *Suppose that none of the simple types in $S = s_1 \dots s_n$ is critical and all of the simple types in $T = t_1 \dots t_m$ are right-only. Then ST has at most one reduction to the empty string. Moreover, if a reduction to the empty string exists, it is computed by Lazy Parsing.*

PROOF: Suppose $R : ST \Rightarrow 1$. Every critical type is a right endpoint and cannot be linked to another critical type. Hence it is linked to some s_i with $i \leq n$.

$$\dots \underbrace{s_{i_1} \dots \dots s_{i_{m-1}} \dots s_{i_m} \dots t_1 t_2 \dots}_{\dots} \dots t_m$$

Omitting these links from R , we obtain a reduction R' of S to the irreducible string $s_{i_1} \dots s_{i_{m-1}} s_{i_m}$. Indeed, the segment between s_{i_m} and t_1 is reduced by R to the empty string, because links do not cross. For the same reason, the segments between two consecutive s_{i_k} 's and the initial segment up to s_{i_1} are reduced to the empty string by R and hence by R' . Now assume $Q : ST \Rightarrow 1$ is another reduction to the empty string. Then its restriction to Q' to S as well is a reduction of S to an irreducible form $s_{j_1} \dots s_{j_{m'}}$. By the Linearity Lemma, S has a unique irreducible form and a unique reduction to this irreducible form. Therefore $Q' = R'$, $m = m'$ and $i_k = j_k$ for $1 \leq k \leq m$. As links do not cross, it follows that $Q = R$. Finally, as S is linear, Lazy Parsing computes the unique reduction $R' : S \Rightarrow s_{i_1} \dots s_{i_m}$. The indices i_1, \dots, i_m are stored in its stack in increasing order. Thus i_m is on the top of the stack after computing at stage n . Reading the type t_1 at the next stage, it will pop i_m and add $\{i_m, n + 1\}$ to the reduction and so on.

We call a string of the form ST in the lemma above *quasi-linear* and remark that it can have several irreducible forms. For example, the empty string and bb^ℓ are two irreducible forms of $bb^\ell bb^\ell bb^r$.

In the next section we relax the restrictive conditions above, motivated by properties of the sample dictionary of Section 3. We also modify the parsing algorithm slightly so that it is complete and linear for the larger class of dictionaries defined by the weaker conditions.

5 Minimal Reductions

Dictionaries of complexity 2 may have type assignments with critical triples. Such strings often have different irreducible forms and even for a given irreducible form there may be different reductions to it. The time necessary to find all possible reductions to the sentence type can increase exponentially with the number of words, and this even if the dictionary lists just one type per word. The best we can attempt therefore is to compute one particular reduction to the sentence type per type assignment, provided there is at least one. In this section we single out the *minimal* reduction always present among all reductions to the sentence type.

The critical triples of our sample dictionary of Section 3 are due to the lexical entries

$$\begin{aligned} \text{which} & : c_s^r c_s \hat{s}^\ell \hat{o}, c_s^r c_s \hat{s}^\ell \hat{\pi}_{3s} \\ \text{and} & : x^r x x^\ell, \end{aligned}$$

A critical type in these entries is followed by its own left adjoint or is in a segment that is followed by its left adjoint. This property is the key to complete linear parsing.

Definition 11. (Guards) A type $T = XCY$ is *guarded* if

- there are no critical types in X nor Y
- every simple type in C is critical
- C^ℓ is the smallest element in its connected component
- $Y = C^\ell Y'$ for some Y' .

If C is not empty, the following segment C^ℓ is called the *guard* of C .

A *dictionary* \mathcal{D} is *guarded* if every type $T \in \mathcal{D}(v)$ is guarded for all v . A *pre-group grammar* is *guarded* if its dictionary is guarded.

Every dictionary of complexity 1 is guarded, because such a dictionary has no critical types. In general, the dictionaries proposed for natural languages so far are guarded or have equivalent guarded dictionaries. At this stage it is not known if guarded grammars are as expressive as the whole class of pregroup grammars.

Our aim is to define a subset of the set of all reductions that is small enough for a slightly amended Lazy Parsing algorithm to compute its members. It also must be big enough so that if a given string has a reduction to the sentence type, one of them is in the subset. We can do this for guarded dictionaries of complexity 2, for which we define the *minimal reductions* below.

Definition 12. (Fans:) Consider a reduction R of $t_1 \dots t_n$. A *fan* of R is a subset of underlinks $\{\{i_p, k\}, \{i_{p-1}, k+1\}, \dots, \{i_1, k+p-1\}\}$ such that the right endpoints form a segment.

$$\dots t_{i_1} \dots \dots t_{i_{p-1}} \dots \dots t_{i_p} \dots t_k t_{k+1} \dots t_{k+p-1} \dots \dots$$

$\underbrace{\hspace{15em}}$

A *fan is critical*, if every right endpoint is a critical type. A *guarded fan* is a critical fan such that $t_{k+p} \dots t_{k+2p-1}$ guards the segment $t_k \dots t_{k+p-1}$ of its right endpoints.

A fan reduces the segments between to left endpoints to the empty type and the same is true for the segment between the last left and the first right endpoint. This follows at once from the following, more general property.

Lemma 13. Fan Lemma *Consider the following underlinks*

$$\dots t_{i_1} \Gamma_1 \dots \dots t_{i_{p-1}} \Gamma_{p-1} \underbrace{t_{i_p} \Theta t_k \Lambda_1 t_{k+1} \dots \Lambda_{p-1} t_{k+p-1}} \dots \dots$$

of a reduction R . Then $R : \Gamma_i \Rightarrow 1$ if and only if $R : \Lambda_{p-i} \Rightarrow 1$ for $1 \leq i \leq p-1$. Moreover, $R : \Theta \Rightarrow 1$. In particular, the segments between two consecutive left endpoints of a fan reduce to the empty string.

PROOF: The last assertion holds in any reduction. Moreover, a link that has one endpoint in Γ_i has the other endpoint either in Γ_i itself or in Λ_{p-i} . Hence $R : \Lambda_{p-i} \Rightarrow 1$ implies $R : \Gamma_i \Rightarrow 1$ and *vice versa*.

Definition 14. (Minimal reductions:) Let $t_1 \dots t_n$ be a string of simple types. The fan

$$\dots t_{j_1} \dots \dots t_{j_{p-1}} \dots \dots \underbrace{t_{j_p} \dots t_k t_{k+1}} \dots t_{k+p-1} \dots$$

is *shorter* than the fan

$$\dots t_{i_1} \dots \dots t_{i_{p-1}} \dots \dots \underbrace{t_{i_p} \dots t_k t_{k+1}} \dots t_{k+p-1} \dots$$

if $i_p < j_1$. A fan is *minimal* if there is no shorter fan with the same right endpoints.

A *reduction* of $t_1 \dots t_n$ to the empty string is *minimal* if all guarded critical fans are the right endpoint of a minimal fan.

In the case where a minimal fan has just one right endpoint, we call it a *minimal link*. Note that a minimal reduction may have non-minimal links. For example, consider the reductions

$$\underbrace{a^\ell a}_a \underbrace{a^\ell a^r}_a \underbrace{a}_a \quad \underbrace{a^\ell a a^\ell a^r}_a \underbrace{a}_a \quad \underbrace{b b^\ell b}_b \underbrace{b^\ell b}_b \underbrace{b^r}_b \quad \underbrace{a^\ell b^\ell b a a^\ell a^r}_b \underbrace{b^r}_b \underbrace{b a}_b \dots$$

The first reduction is minimal because all its links are minimal. The second is not minimal, it has a guarded non minimal link. The third reduction is also minimal, because its only critical type b^r is not guarded and therefore is not required to be minimal. The fourth is again minimal, even though the link through a^r is not.

Theorem 15 (Completeness of Minimal Reductions). *Assume that \mathcal{D} is a guarded dictionary of complexity 2. Then for every type assignment $X_i \in \mathcal{D}(v_i)$ such that $X_1 \dots X_n \rightarrow \mathbf{s}$ there is a minimal reduction of $X_1 \dots X_n$ to \mathbf{s} .*

PROOF: We must show that whenever $R : X_1 \dots X_n \mathbf{s}^r = t_1 \dots t_m \Rightarrow 1$ is a reduction then there is a minimal reduction M such that $M : t_1 \dots t_m \Rightarrow 1$. We use induction on the number of critical types in $t_1 \dots t_m$.

The property is trivially true if there is no critical type. Otherwise, let t_k be the leftmost critical type. Note that the string $t_1 \dots t_{k-1}$ is linear. As the dictionary is guarded, t_k is either the last type in the string or is followed by another critical type or t_{k+1} is the guard of t_k . In the first case the reduction R is trivially minimal. Now assume that t_{k+1} is the guard of t_k .

Consider two cases.

Case (i):

The link $\{i, k\}$ is minimal.

Then the restriction R' of R to $\{i, \dots, k\}$ is a reduction $R' : t_i \dots t_k \Rightarrow 1$. Omit the segment $t_i \dots t_k$ from $t_1 \dots t_m$ and the links with an endpoint in $\{i, \dots, k\}$ from R , obtaining the string $t_1 \dots t_{i-1} t_{k+1} \dots t_m$ and a reduction $R_1 : t_1 \dots t_{i-1} t_{k+1} \dots t_m \Rightarrow 1$. As the induction hypothesis applies to R_1 , there

is a minimal reduction $M^* : t_1 \dots t_{i-1} t_{k+1} \dots t_m \Rightarrow 1$. Inserting the segment $t_i \dots t_k$ with the underlinks of R' into M^* we obtain a minimal reduction of $t_1 \dots t_m$ to the empty type.

Case (ii): The link $\{i, k\}$ is not minimal and t_{k+1} guards t_k .

Let j be the rightmost position $i < j < k$ such that there is a reduction R^* satisfying

$$\{j, k\} \in R^* \quad (\text{II})$$

$$t_j t_k \rightarrow 1 \quad (\text{III})$$

$$R^* : \Theta = t_{j+1} \dots t_{k-1} \Rightarrow 1 \quad (\text{IV})$$

It suffices to construct a reduction $R' : t_1 \dots t_m \Rightarrow 1$ linking j and k and conclude by Case(i).

First we remark that R coincides with R^* on Θ . Indeed, $t_{i+1} \dots t_{k-1}$ is linear and both R and R^* link a type in Θ to a type in $t_{i+1} \dots t_{k-1}$. Therefore they coincide in Θ by the Linearity Lemma 7. Hence

$$R : \Theta = t_{j+1} \dots t_{k-1} \Rightarrow 1. \quad (\text{V})$$

It follows that R links j to a position h between i and j . Hence

$$\{h, j\} \in R, \quad h < j, \quad t_h t_j \rightarrow 1 \quad (\text{VI})$$

$$R : \Delta = t_{h+1} \dots t_{j-1} \Rightarrow 1. \quad (\text{VII})$$

Then R has the underlinks

$$\underbrace{t_i \Gamma t_h \Delta t_j \Theta t_k}_{\text{underlink}}. \quad (\text{VIII})$$

As R links each position in Θ to another position in Θ , the same holds for Γ , i.e.

$$R : \Gamma = t_{i+1} \dots t_{h-1} \Rightarrow 1. \quad (\text{IX})$$

Let l be the position linked to $k+1$ in R . Note that either $l > k+1$ or $l < i$. If $l > k+1$, i.e. if

$$R \quad \dots \underbrace{t_i \Gamma t_h \Delta t_j \Theta t_k}_{\text{underlink}} t_{k+1} \dots t_l \dots, \quad (\text{X})$$

construct a new reduction $R' : t_1 \dots t_m \Rightarrow 1$ by omitting the links $\{i, k\}$, $\{h, j\}$ and $\{k+1, l\}$ from R and adding the links $\{j, k\}$, $\{h, k+1\}$ and $\{i, l\}$ instead. The other links remain unchanged. Then R' looks like this

$$R' \quad \dots \underbrace{t_i \Gamma t_h \Delta t_j \Theta t_k t_{k+1} \dots t_l}_{\text{underlink}} \dots \quad (\text{XI})$$

If $l < i$ i.e. if

$$R \quad \dots \underbrace{t_l \dots t_i \Gamma t_h \Delta t_j \Theta t_k}_{\text{underlink}} t_{k+1} \dots, \quad (\text{XII})$$

let again R' is obtained from R by omitting the links $\{i, k\}$, $\{h, j\}$ and $\{k + 1, l\}$ from R and adding the links $\{j, k\}$, $\{h, k + 1\}$ and $\{i, l\}$ instead. Now, R' looks like this

$$R' \quad \dots \underline{t_l \dots t_j} \Gamma \underline{t_h \Delta t_j \Theta t_k} t_{k+1} \dots \quad (\text{XIII})$$

To see that $R' : t_1 \dots t_m \Rightarrow 1$, note that the geographical conditions are satisfied because of V, VII and IX. Moreover, the algebraic conditions obviously hold for the unchanged links and for the minimal link $\{j, k\}$. They also hold for the new links. Indeed, $t_{k+1} = t_k^\ell$ is the smallest element in its connected component by assumption. Hence whenever $tt_k \rightarrow 1$ then $t = t_{k+1}$.

Remains the case where t_{k+1} is another critical type, i.e. where the leftmost critical fan has several right endpoints. We only consider the case where R is not minimal. Let R^* be a reduction that defines a shorter fan than R .

$$R^* \quad \dots \dots a \Lambda \underbrace{b \dots b^r a^r}_{\text{---}} ab \dots$$

Let Λ_q be the intermediary segment of any two successive left endpoints of R^* . Using linearity of the string before the first critical type we show that R coincides with R^* on Λ_q and cancels the left endpoints of R^* to the right of its own left endpoints

$$R \quad \dots a \dots b \dots \underline{b^\ell \Gamma a^\ell \dots a \Lambda b} \dots b^r a^r ab \dots$$

From this follows by the Fan Lemma,

$$R : \Lambda_q \Rightarrow 1, \quad R : \Gamma_q \Rightarrow 1$$

Now we construct R' as above. This ends the proof.

6 Minimal Parsing

Recall that a string with n critical types may have up to 2^n different reductions. Hence looking for only one of them should diminish the work considerably. In fact, Lazy Parsing can be amended to find such a minimal reduction keeping its run-time still linear. We formulate the relevant properties of Lazy Parsing in the next two lemmas. For a fixed string of simple types $T = t_1 \dots t_n$, the stages of Lazy Parsing identify with the positions $\{0, 1, \dots, n\}$, where 0 stand for the initial stage. Hence, Lazy Parsing processes t_i at stage i .

Lemma 16. *Assume that the reduction $R : T = t_1 \dots t_n \Rightarrow 1$ includes the critical fan*

$$\dots \underline{t_{j_1} \dots t_{j_{p-1}} \dots t_{j_p} \dots t_k t_{k+1} \dots t_{k+p-1}} \dots$$

If T is of complexity 2 then $t_{j_1} \dots t_{j_p}$ is an irreducible form of $t_{j_1} \dots t_{k-1}$.

PROOF: All we must show is that $t_{j_q} t_{j_{q+1}} \not\rightarrow 1$ for $1 \leq q \leq p-1$. Let $a^{(z)} = t_{j_q}$ and $b^{(u)} = t_{j_{q+1}}$. As R links $a^{(z)}$ to t_{p+q-1} the latter has the form $t_{p+q-1} = a'^{(z+1)}$ for an appropriate a' in the connected component of a . Similarly, $t_{p+q} = b'^{(u+1)}$ for some b' in the connected component of b . If $t_{j_q} t_{j_{q+1}} \rightarrow 1$, then $u = z+1$. Moreover, a, a', b and b' belong to the same connected component and $a^{(z)}, a'^{(z+1)}, b^{(z+1)}$ and $b'^{(z+2)}$ occur in T . Then $a'^{(z+1)}$ cannot be a critical type because T has complexity 2

Lemma 17. *Assume that $R : t_1 \dots t_n \Rightarrow 1$, that t_k is the leftmost critical type with corresponding link $\{j, k\}$ in R . Let i be the top of the stack constructed by Lazy Parsing when processing t_{k-1} . Then $i \leq j \leq k-1$.*

In particular, if $i < j$, then there are l and m such that $i < l < j, k < m$, Lazy Parsing links t_l to t_j , R links t_l to t_m and the latter is not a critical type.

PROOF: Let $R : t_1 \dots t_n \Rightarrow 1$, i, j and k be as in the hypothesis of the Lemma. As t_k is a critical type in a dictionary of complexity 2, it can only be a right endpoint, hence $j \leq k$. To see that $i \leq j$ assume that this was not the case, i.e. that

$$\dots \underline{t_j \dots t_i \dots t_k} \dots$$

Hence R links i to some position l strictly between j and k . We remark first that $i < l$ is impossible. Indeed, Lazy Parsing reduces the segment $t_{i+1} \dots t_{k-1}$ to the empty string by property (2) of Theorem (3). Hence it would link l to some p strictly between i and k

$$\dots t_i \dots \underbrace{t_p \dots t_l \dots t_k} \dots,$$

contradicting the uniqueness of links in the linear segment $t_i \dots t_{k-1}$. Therefore R must link t_i to t_l for some $l < i$.

$$\dots \underline{t_j \dots t_l \dots t_i \dots t_k} \dots$$

Next, we remark that l is put on top of the stack while processing t_l , because t_l does not contract with any type on its left. For if it did, t_i would be a critical type, contradicting the assumption. When processing the string from t_l toward the right, Lazy Parsing cannot pop l from the stack before reaching t_i . Indeed, this would imply that t_l is endpoint of two different links in a linear string, contradicting Lemma (7). It follows that Lazy Parsing computes an irreducible form of $t_l \dots t_i$ different from the empty string, contradicting lemma (8). Therefore $j < i$ is impossible and so $i \leq j \leq k-1$.

Consider the case $i < j$. Recall Property (2) of Theorem (3) which says that Lazy Parsing reduces $t_{i+1} \dots t_{k-1}$ to the empty string. Hence it links j to some position l with $i < l < k$. Moreover, we have $l < j$ because t_k is the first critical type. Hence

$$\dots t_i \dots \underbrace{t_l \dots t_j \dots t_k} \dots,$$

where the underbraces designate links of Lazy Parsing. Note that t_l is left-only, because the dictionary has complexity 2. Therefore R links l to some $m > l$. Moreover, if we had $m \leq k-1$, then l would be linked to two different positions in the linear string $t_1 \dots t_{k-1}$ which is impossible. Hence $k+1 \leq m$.

Lemma 18. *Assume the notations of the preceding lemma and that t_k is the leftmost critical type of a fan*

$$\dots t_{j_1} \dots \underbrace{\dots t_{j_{p-1}} \dots t_{j_p} \dots t_k t_{k+1} \dots t_{k+p-1}} \dots$$

If $j_p > i$ then $j_q > i$ for $1 \leq q \leq p$. Moreover, there are $j_1 > l_1 > \dots > l_p > i$ such that Lazy Parsing links t_{l_q} to t_{j_q} .

PROOF: Use induction on the number of critical types p . The case $p = 1$ is the preceding lemma. For the induction step, omit the segment $t_{j_p} \dots t_k$ from $t_1 \dots t_n$ and the corresponding links from R . Apply the induction hypothesis to this string to obtain $l_{p-1} < \dots < l_1$ and the corresponding links of Lazy Parsing

$$\dots t_i \dots t_{l_{p-1}} \dots \underbrace{t_{l_1} \dots t_{j_1}} \dots \underbrace{t_{j_{p-1}} \Gamma_p t_{j_p} \dots t_k t_{k+1}} \dots t_{k+p-1} \dots$$

Use the usual argument to show that Lazy Parsing links t_{l_p} to t_{j_p} for some l_p between i and j_p

$$\dots t_i \dots \underbrace{t_{l_p} \dots t_{j_p}} \dots t_k \dots$$

We have $l_p < l_{p-1}$ or $j_{p-1} < l_p$, for Lazy Parsing links l_{p-1} to j_{p-1} . We remark that the latter is impossible. Indeed, if t_{l_p} was inside of Γ_p it would be linked by R to some type before t_{j_p} contradicting the uniqueness of links in the linear string Γ_p . Therefore $l_p < l_{p-1}$.

Minimal Parsing

Let $t_1 \dots t_n = T_1 \dots T_m s^r$ be a type assignment and assume that the dictionary is guarded. Then the first critical type is the leftmost type of a segment CC^ℓ within the type of some word. Hence the length of C is a constant depending only on the dictionary.

We amend Lazy Parsing by back-tracking when arriving at the first critical type t_k with i on the top of the stack. Instead of processing t_k , we process the string $t_i \dots t_{k-1}$ backward, starting at $k-1$, but no further than $i+1$ included. We construct a separate stack and set of links for the backward parsing. At each position, we compare the computed irreducible form, say $t_{j_1} \dots t_{j_q}$, with the string C^ℓ . If and when it satisfies $C^\ell = t_{j_1} \dots t_{j_p}$, we stop and update Lazy Parsing following the construction of a minimal reduction in Theorem (15). We omit the links $\{l_q, j_q\}$ from the reduction computed so far and add the links $\{j_q, k+q-1\}$ and $\{l_q, k+p-1+q\}$ instead. The stack remains unchanged. Then we continue processing forward with Lazy Parsing from t_{k+2p} on. If on the contrary, the irreducible form computed by backtracking never becomes equal to C^ℓ for $k-1 \geq q \geq i+1$, we continue with forward Lazy Parsing from t_k on,

with the stack and set of links as they were before backtracking. If this procedure links the critical types successfully on the left, the corresponding segment $t_{j_1} \dots t_{k+p-1}$ will be omitted in later backtracks. Hence every simple type in the string is processed at most twice.

It follows from Lemmas (17) and (18) that Minimal Parsing is complete if the dictionary is guarded and has complexity 2. Moreover, it computes a reduction to the sentence time or fails if there is none, and does so in time proportional to the length of the string $T_1 \dots T_m$.

We can use Minimal Parsing for recognizing ‘losing’ type assignments with no reduction to the sentence type, when processing a word that introduces a critical type. If $k + q$ is pushed onto the stack for some $0 \leq q \leq p - 1$, then $t_1 \dots t_n$ has no reduction to the empty string. Indeed, if Minimal Parsing does not find a link for the critical type t_{k+q} then there is no such link by Theorem (15) and Lemmas (17) and (18). As t_{k+q} cannot be cancelled from the right, it remains in every irreducible form of $t_1 \dots t_n$. Note that we do not even need to know what will come after the critical segment and its guard. We only need to know the critical segment C so that we can test the irreducible form during backtracking for equality with C^ℓ .

7 Conclusion

The conditions which make the Minimal Parsing algorithm linear apply to a large class of pregroup grammars. Indeed, a pregroup grammar has a finite dictionary and therefore is strongly equivalent to one of complexity 2. Practice shows that guarded dictionaries cover quite expressive natural language fragments. However, Minimal Parsing depends on the selected type assignment. Future work must investigate dictionaries for which the criterion for recognizing losing type assignments during processing lowers the number of processed type assignments sufficiently. The present work is only a first step toward the claim that pregroup grammars can provide natural language processing with linear algorithms.

References

- [Buszkowski] Wojciech Buszkowski, Lambek Grammars based on pregroups, in: P. de Groote et al., editors, Logical Aspects of Computational Linguistics, LNAI 2099, Springer, 2001
- [CYK] David Younger, Recognition and Parsing of Context-Free Languages in Time n^3 , Information and Control, 10:2, 1967
- [Degeilh-Preller] Sylvain Degeilh, Anne Preller, Efficiency of Pregroups and the French noun phrase, Journal of Language, Logic and Information, Springer, Vol. 14, Number 4, pp. 423-444, 2005

- [Earley] Jay Earley, An efficient context-free parsing algorithm, Communications of the AMC, Volume 13, Number 2, pp 94-102, 1970
- [Gazdar] Gerald Gazdar, Unbounded Dependency and Coordinate Structure, in: The Formal Complexity of Natural Language, Walter Salvitch, ed., Reidel Publishing Company, pp. 183-226, 1987
- [Lambek 99] Joachim Lambek, Type Grammar revisited, in: A. Lecomte et al., editors, Logical Aspects of Computational Linguistics, Springer LNAI 1582, pp.1 -27, 1999
- [Lambek 04] Joachim Lambek, A computational algebraic approach to English grammar, Syntax 7:2, pp. 128-147, 2004
- [Oehrle] Richard Oehrle, A parsing algorithm for pregroup grammars, in: Proceedings of Categorical Grammars 2004, Montpellier France, pp.59-75, 2004
- [Preller-Lambek] Anne Preller, Joachim Lambek (2007), *Free compact 2-categories*, Mathematical Structures for Computer Sciences, vol. 17, pp.309-340, Cambridge University Press. doi:10.1017/S0960129506005901
- [Preller 07] Anne Preller (2007), *Toward Discourse Representation Via Pregroup Grammars*, JoLLI, Vol.16, pp. 173-194. doi:10.1007/s10849-006-9033-y