LinPP: a Python-friendly algorithm for Linear Pregroup Parsing

Irene Rizzo
University of Oxford / Oxford
irene.rizzo@cs.ox.ac.uk

Abstract

We define a linear pregroup parser, by applying some key modifications to the *minimal parser* defined in (Preller, 2007a). These include handling words as separate blocks, and thus respecting their syntactic role in the sentence. We prove correctness of our algorithm with respect to parsing sentences in a subclass of pregroup grammars. The algorithm was specifically designed for a seamless implementation in Python. This facilitates its integration within the *DisCopy* module for QNLP and vastly increases the applicability of pregroup grammars to parsing real-world text data.

1 Introduction

Pregroup grammars (PG), firstly introduced by J. Lambek in (Lambek, 1997), are becoming popular tools for modelling syntactic structures of natural language. In compositional models of meaning, such as *DisCoCat* (Coecke et al., 2010) and *DisCoCirc* (Coecke, 2019), grammatical composition is used to build sentence meanings from words meanings. Pregroup types mediate this composition by indicating how words connect to each other, according to their grammatical role in the sentence. In DisCoCat compositional sentence embeddings are represented diagrammatically; these are used as a language model for QNLP, by translating diagrams into quantum circuits via the Z-X formalism (Zeng and Coecke, 2016; Coecke et al., 2020a,b; Meichanetzidis et al., 2020b,a). *DisCopy*, a Python implementation of most elements of DisCoCat, is due to Giovanni Defelice, Alexis Toumi and Bob Coecke (Defelice et al., 2020).

An essential ingredient for a full implementation of the DisCoCat model, as well as any syntactic model based on pregroups, is a correct and efficient pregroup parser. Pregroup grammars are weakly equivalent to context-free grammars (Buszkowski, 2009). Thus, general pregroup parsers based on this equivalence are poly-time, see e.g. (Earley, 1970). Examples of cubic pregroup parsers exist by Preller (Degeilh and Preller, 2005) and Moroz (Moroz, 2009b) (Moroz, 2009a). The latter have been implemented in Python and Java. A faster *Minimal Parsing* algorithm, with linear computational time, was theorised by Anne Preller in (Preller, 2007a). This parser is correct for the subclass of pregroup grammars characterised by *guarded dictionaries*. The notion of *guarded* is defined by Preller to identify dictionaries, whose *criticalities* satisfy certain properties (Preller, 2007a). In this paper we define *LinPP*, a new linear pregroup parser, obtained generalising Preller's definition of guards and applying some key modifications to the Minimal Parsing algorithm. *LinPP* was specifically designed with the aim of a Python implementation. Such implementation is currently being integrated in the DisCopy package (github:*oxford-quantum-group/discopy*).

The need for a linear pregroup parser originated from the goal of constructing a grammar inference machine learning model for pregroup types, i.e. a Pregroup Tagger. training and evaluation of such model is likely to involve parsing of several thousand sentences. Thus, LinPP will positively affect the overall efficiency and performance of the Tagger. The Tagger will enable us to process real world data and test the DisCoCat pregroup model against the state-of-the-art with respect to extensive tasks involving real-world language data.

2 Pregroup Grammars

We recall the concepts of *monoid*, *preordered monoid* and *pregroup*.

Definition 2.1. A monoid $\langle P, \bullet, 1\rangle$ is a set P together with binary operation $\bullet$ and an element 1, such that

$$(x \bullet y) \bullet z = x \bullet (y \bullet z) \tag{1}$$

Proceedings of the 2021 Workshop on Semantic Spaces at the Intersection of NLP, Physics, and Cognitive Science, pages 12–19

$$x \bullet 1 = x = 1 \bullet x \tag{2}$$

for any $x, y \in P$. We refer to $\bullet$ as *monoidal product*, and we often omit it, by simply writing xy in place of of $x \bullet y$.

Definition 2.2. A preordered monoid is a monoid together with a *reflexive transitive* relation $P \to P$ such that:

$$x \to y \implies uxv \to uyv \tag{3}$$

Definition 2.3. A **pregroup** is a preordered monoid $\langle P, \bullet, 1 \rangle$, in which every object x has a left and a right *adjoint*, respectively written as x^l and x^r, such that:

$$\begin{aligned} &\text{contraction rules} \quad x^l x \to 1; \quad xx^r \to 1 \\ &\text{expansion rules} \quad 1 \to x^r x; \quad 1 \to xx^l \end{aligned}$$

Adjoints are unique for each object.

In the context of natural language, pregroups are used to model grammatical types. This approach was pioneered by J. Lambek, who introduced the notion of *Pregroup Grammars* (Lambek, 1997). These grammars are constructed over a set of *basic types*, which represent basic grammatical roles. For example, $\{n, s\}$ is a set consisting of the *noun* type and the *sentence* type.

Definition 2.4. Let B be a set of *basic* types. The free pregroup over B, written P_B, is the free pregroup generated by the set $B \cup \Sigma$, where Σ is the set of iterated adjonts of the types in B.

In order to easily write iterated adjoints, we define the following notation.

Definition 2.5. Given a basic type t, we write t^{l^n} to indicate its n-fold left adjoint, and t^{r^n} for its n-fold right adjoint.E.g. we write t^{l^2} to indicate $(t^l)^l$.

Thanks to the uniqueness of pregroup adjoints we can mix the right and left notation. E.g. $(t^{r^2})^l$ is simply t^r. We write $t^{l^0} = t = t^{r^0}$. We now define pregroup grammars, following the notation of (Shiebler et al., 2020).

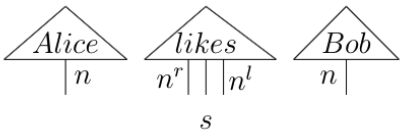
Definition 2.6. A **pregroup grammar** is a tuple $PG = \{V, B, \mathbb{D}, P_B, s\}$ where:

1. V is the *vocabulary*, i.e. a set of words.
2. B is a set of basic grammatical types.
3. P_B is the free pregroup over B.
4. $\mathbb{D} \subset V \times P_B$ is the *dictionary*, i.e it contains correspondences between words and their assigned grammatical types.
5. $s \in P_B$ is a basic type indicating the *sentence type*.

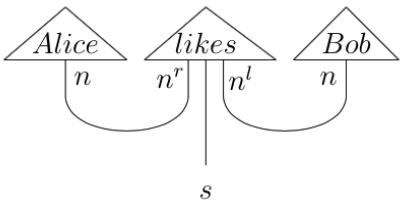
Example 2.7. Consider the grammar given by $V = \{Alice, loves, Bob\}$, $B = \{n, s\}$ and a dictionary with the following type assignments:

$$\mathbb{D} = \{(Alice, n), (Bob, n), (loves, n^r s n^l)\}$$

Note that the grammatical types for *Alice* and *Bob* are so-called *simple* types, i.e basic types or their adjoints. On the other hand, the type of the transitive verb is a monoidal product. The type of this verb encodes the recipe for creating a sentence: it says *give me a noun type on the left and a noun type on the right and I will output a sentence type*. In other words, by applying iterated contraction rules on $nn^r sn^l n$ we obtain the type s. Diagrammatically we represent the string as



Then, after applying the contraction rules, we obtain a sentence diagram:



This diagram is used to embed the sentence meaning. This framework - introduced by Coecke et al. in 2010 - is referred to as *DisCoCat* and provides a mean to equip distributional semantics with compositionality. The composition is mediated by the sentence's pregroup contractions, as seen in the example above. (Coecke et al., 2010).

The iterated application of contraction rules yields a *reduction*.

Definition 2.8. Let $S := t_1....t_n$ be a string of simple types, and let $T_S := t_{j_1}....t_{j_p}$ with $j_i \in [1, n]$ for all i. We say that $R : S \to T_S$ is a

reduction if R is obtained by iterating contraction rules only. We say that T_S is a reduced form of S. If T_S cannot be contracted any further, we say that it is **irreducible** and we often write $R : S \implies T_S$. Note that neither reductions nor irreducible forms are unique, as often we are presented with different options on where to apply contraction rules.

In the context of pregroup grammars, we are interested in reducing strings to the sentence type s, whenever this is possible. Thus, we give such reduction a special name (Shiebler et al., 2020):

Definition 2.9. a reduction $R : S \implies T_S$ is called a **parsing** of S, if T_s is the simple type s. A string S is a *sentence* if there exists a parsing.

Often, we want to keep track of the types as they get parsed:

Definition 2.10. The **set of reductions** of $R : S \to T_S$ is a set containing index pairs $\{i, j\}$ such that $t_i t_j$ is the domain of a contraction in R. These pairs are referred to as *underlinks*, or *links* (Preller, 2007a).

3 Linear vs critical

We now discuss critical and linear types in a pregroup grammar. We first need to introduce the notion of *complexity* (Preller, 2007a, Definition 5) [Preller].

Definition 3.1. A pregoup grammar with dictionary $\mathbb{D}$ has **complexity** k if, for every type $t \in \mathbb{D}$, any left (right) adjoint t^{l^n} (t^{r^n}) in $\mathbb{D}$ is such that $n < k$.

Complexity 1 indicates a trivial grammar that contains only basic types (no adjoints). Complexity 2 allows for dictionaries containing at most basic types and their 1-fold left and right adjoints, e.g. n^l and n^r. As proven in (Preller, 2007b), every pregroup grammar is *strongly equivalent* to a pregroup grammar with complexity 2. This means that the subclass of complexity 2 pregroup grammars has the same expressive power of the whole class of pregroup grammars.

We now introduce critical types (Preller, 2007a).

Definition 3.2. A type c is **critical**, if there exists types $a, b \in \mathbb{D}$ such that $ab \to 1$ and $bc \to 1$. A type is *linear* if it is not critical.

We say that a grammar is linear if all types in the dictionary are linear types. Given a string from a linear grammar, its reduction links are unique (Preller, 2007a, Lemma 7). In fact, a very simple algorithm can be used to determine whether a linear string is a sentence or not.

3.1 Lazy Parsing

The Lazy Parsing algorithm produces parsing for all linear sentences.

Definition 3.3. Consider a linear string S. Let St be an initially empty stack, and R an initially empty set of reductions. The Lazy Parsing algorithm reduces the string as follows:

1. The first type in S is read and added to St.
2. Any following type t_n is read. Letting t_i indicate the top of the stack St up until then, if $t_i t_n \to 1$ then St is popped and the link is added to R. Otherwise t_n is added to St and R remains unchanged.

By (Preller, 2007a, Lemma8, Lemma9) Lazy Parsing reduces a linear string to its *unique* irreducible form, thus a linear string is a sentence if and only if the Lazy Parsing reduces it to s. Unfortunately linear pregroup grammars do not hold a lot of expressive power, and criticalities are immediately encountered when processing slightly more complex sentences than 'subject + verb + object'. Thus, defining parsing algorithms that can parse a larger class of pregroup grammars becomes essential.

3.2 Guards

In order to discuss new parsing algorithms in the next sections, we introduce some useful notions.

Definition 3.4. Given a reduction R, a subset of *nested* links is a called a **fan** if the right endpoints of the links form a segment in the string. A fan is critical if the right endpoints are critical types (Preller, 2007a).

Below, we define *guards*, reformulating the notion introduced by Preller in (Preller, 2007a).

Definition 3.5. Let us consider a string $S := t_1 .... t_b = X t_p Y$, containing a critical type t_p. Let S reduce to 1. We say that t_b is a **guard** of t_p in S if the following conditions are satisfied:

1. X contains only linear types and there exists a reduction $R : X \implies 1$.
2. There exists a link $\{j, k\}$ of R such that $t_k t_p \to 1$ and $t_j t_b \to 1$ are contractions.

3. There exist subreductions $R1, R_2 \subset R$ such that $R_1 : t_{k+1}..t_{p-1} \implies 1$ and $R_2 : t_1...t_{j-1} \implies 1$.

4. There exists a reduction $R_y : Y \implies t_b$.

If such guard exists, we say that the critical type is *guarded* and we say that $\{j, b\}$ is a *guarding link* for the critical type.

Let us adapt this definition to critical fans.

Definition 3.6. Let us consider the segment $S := t_1.....t_n = XT_cY$. Let us assume there exists a reduction $S \implies 1$, that contains a critical fan with right end points $t_p....t_{p+q} =: T_c$. We say that the fan is guarded in S if:

1. X is linear and there exists a reduction $R : X \implies 1$.

2. There exist links $\{j_i, k_i\} \in R$, for $i \in [p, p+q]$, with $k_p > ... > k_{p+q}$, $j_p < ... < j_{p+q}$ and $t_{k_{p+q}}...t_{k_p}T_c \implies 1$.

3. The segments $t_1...t_{j_p}$ and $t_{k_p+1}..t_{p-1}$, as well as the ones in between each t_k or t_j and the next ones, have reductions to 1.

4. There exists a reduction $R_y : Y \implies T_c^l$.

3.3 Critical types in complexity 2 grammars

Critical types are particularly well behaved in dictionaries of complexity 2, as they are exactly the right adjoints t^r of basic types t. We recall the following results from (Preller, 2007a, Lemma 17 & 18). We assume complexity 2 throughout.

Lemma 3.7. Let $R : t_1...t_m \implies 1$. Let t_p be the leftmost critical type in the string and let R link $\{k, p\}$. Let t_i be the top of the stack produced by Lazy Parsing, then $i \leq k$. Moreover, if $k > i$, there are j, b with $i < j < k$ and $b > p$, such that Lazy Parsing links $\{j, k\}$ and R links $\{j, b\}$.

Corollary 3.8. *Let t_p be the leftmost critical type of a sentence S. With i as above, if $t_i t_p$ reduce to the empty type, then all reductions to type s will link $\{i, p\}$.*

We prove the following result.

Lemma 3.9. Let $S := s_1....s_n$ be a string with $m \geq 2$ critical types. Let them all be guarded. Let s_p be a critical type, and let s_q be the next one. Let s_{b_p} and s_{b_q} be their guards respectively. Assume the notation of the previous definitions. Then, either $j_q > p$ and $b_q < b_p$, or $j_q > b_p$.

Proof. By assumption, s_p is guarded, and by definition of guard, the segment $s_{p+1}....s_{b_p-1}$ reduces to the empty type. For the sake of contradiction, assume $j_q < p$. Then, because crossings are not allowed, we must have $j_q < k_p$. Since j_q is a left adjoint of a basic type, it can only reduce on its right, and we have $k_q < k_p$. However, the segment Y_p does contain s_q, and does not contain its reduction s_{k_q}, thus Y_p cannot reduce to type s_{b_p}, which is a contradiction. Thus $j_q > p$, and to avoid crossings, it is either $j_q > p$ and $b_q < b_p$ or $j_q > b_p$. □

The lemmas above also hold for guarded critical fans.

4 $MinPP$: Minimal Parsing Algorithm

In this section we define $MiniPP$, a minimal parsing algorithm complexity 2 pregroup grammars.

MinPP pseudo-code. Let $sentence : t_1....t_m$ be a string of types from a dictionary with complexity 2. We associate each processing step of the algorithm with a stage S_n. Let S_0 be the initial stage, and $S_n := \{a, n\}$ with $n \geq 1$ be the stage processing the type a in position n. Let R_n and St_n be respectively the set of reductions and the reduced stack at stage S_n. Let us write $\top(St_n)$ for the function returning the top element of the stack at stage n and $pop(St_n)$ for the function popping the stack. The steps of the algorithms are defined as follows. At stage S_0, we have $R_0 = \emptyset$ and $St_0 = \emptyset$. At stage S_1, $R_1 = \emptyset$ and $St_1 = t_1$. At stages S_n, $n > 1$, let $t_i = \top(St_{n-1})$. We define the following cases.

- If $t_i t_n \to 1$:

$$St_n = pop(St_{n-1})$$
$$R_n = R_{n-1} \cup \{i, n\}$$

- Elif t_n is linear:

$$St_n = St_{n-1} + t_n$$
$$R_n = R_{n-1}$$

- Else (t_n is **critical**):

  1. **while** types are critical read $sentence$ forward starting from t_n and store read types. Let $T^r := t_n...t_{n+v}$, $v \geq 0$, be the segment of stored types.

2. Create a new empty stack St_{back}. Process $sentence$ backward, starting from T^r and not reading further than t_{i+1}.
3. If St_{back} is never found empty, set $St_n = St_{n-1} + T^r$, $R_n = R_{n-1}$ and move to stage S_{n+v+1} i.e. the first type after the critical fan. If instead St_{back} becomes empty, proceed as follows.
4. St_{back} being empty means that T^r was reduced with some types T. By construction, T had been initially reduced with some types T^l by the forward process. Set $St_n = St_{n-1} + T^l$. Write $R_{T_{prec}}$ for the set of links that originally reduced T^lT. Write R_T for the set of links for the TT^r reduction, as found by the backward process. Set $R_n = (R_{n-1} \cup R_T)/R_{T_prec}$. Move to the next stage.

4.1 Formal Verification

In this section we prove the correctness of $MinPP$ with respect to reducing strings to an irreducible form, given some restrictions on the grammar. First we prove that $MinPP$ is a **sound** and **terminating** parsing algorithm for complexity 2 pregroup grammars. Then, we prove that it is also **correct** with respect to a subclass of complexity 2 pregroup grammars identified bt certain restrictions.

Theorem 4.1. *Let str be a string of types from a complexity 2 pregroup grammar. If we feed str to $MinPP$, then:*

1. ***Termination**: $MinPP$ eventually halts.*
2. ***Completeness**: If str is a sentence, $MinPP$ reduces str to sentence type s.*
3. ***Soundness**: If str is not a sentence, then $MinPP$ will reduce it to an irreducible form different from s.*

Proof. Let t_i always indicate the top of the stack.
Termination. Let us consider strings of finitely many types. We prove that at each stage the computation is finite, and that there are a finite number of stages. A stage S_n is completed once its corresponding stack St_n and set of reductions R_n is computed. If t_n is linear, updating St_n and R_n only involves two finite computations: checking whether $t_it_n \to 1$ (done via a terminating truth value function), and popping or adding t_n to the stack. In the case of t_n being critical, if $t_it_n \to 1$, this is handled like in the linear case. Else, the following computations are involved: first, the algorithm will read forward to identify a critical fan. This will halt when either reading the last critical type of the fan, or the last type in the string. Then the string is processed backward. This computation involves finite steps as in the forward case, and halts when reading t_i or the first type in the string, or when the stack is empty. The next computations involve updating the stack and reduction sets via finite functions. This proves that each step of the process is finite and that $MinPP$ terminates.

Soundness. We prove it by induction on the number of critical fans.
Base Case
Consider a string with one critical fan with right endpoints T^r, and assume it is not a sentence. The case in which the fan reduces with the stack is trivial, so we assume otherwise. We have two cases:
 # 1: Let T^r have a left reduction T. Assuming the notation above, consider segments $\theta_{prec}, \theta, \theta_{post}$. θ reduces to the empty type. So we must have $\theta_{prec}\theta_{post} \to C$, with $C \neq s$. Since this string is linear $MinPP$ will reduce the full string to C.
 # 2: Assume T^r doesn't have a left reduction. Then the backward stack will not become empty, and once the backward parsing will reach t_i, $MinPP$ will add T^r to the forward stack. At this stage, the remaining string will be CT^rD with C possibly empty. D is linear and cannot contain right reductions for T^r since the complexity is 2. Thus $MinPP$ will reduce it by Lazy Parsing to its unique irreducible form $T^rU \neq s$.
Inductive Hypothesis
Assume that $MinPP$ reduces any non-sentences to to an irreducible form different from s, given that the string has no more than m critical fans.
Inductive Step
We consider a string with $m + 1$ critical fans, and no reduction to the sentence type.
 # 1: Assume the notation above and let T^r have left reductions. Then, we remove the segment θ. $MinPP : \theta \implies 1$. The remaining string has m critical fans and no reduction to sentence type, so by induction hypothesis, $MinPP$ won't reduce it to the sentence type.
 # 2: Assume that T^r has no left adjoints in the string. Then, $MinPP$ will add T^r to the to the top of the forward stack. The remaining string

to process is CT^rD, with C linear, irreducible and possibly empty, and D containing m critical fans. Thus, $MinPP$ will correctly parse D to its irreducible form, by inductive hypothesis or by proof of completeness (depending on whether D has a reduction to s or not). Therefore $MinPP$ will reduce CT^rD to an irreducible form, that must be different from s since T^r cannot contain s and cannot reduce further. □

In order to prove completeness we need to restrict our grammars further.

Theorem 4.2. *Let str be a string of types from a complexity 2 pregroup grammar. Let also assume that all critical fans are guarded or their critical types contract with the top of the stack of the corresponding stages. If we feed str to $MinPP$, then:* ***(Completeness)*** *If str is a sentence, $MinPP$ reduces str to sentence type s.*

Proof. We prove it by induction on the number of critical fans.

Base case

Let us consider a sentence with one critical fan, with right-end points $T^r := t_p....t_{p+n}$. At stage S_p, we have two cases: **# 1:** Let $t_it_p \to 1$. Then, by 3.8 all reductions of the string to the sentence type will link i, p. Since links cannot cross, we have $k_q < i$ for all q. Thus all critical types are linked to types in the stack. Thus, their links are unique and will be reduced by Lazy Parsing. By assumption, all types other than the critical fan are linear, thus their links are unique. Thus, Lazy Parsing will correctly reduce this sentence, and, by construction, so will $MinPP$.

 # 2: Let $t_it_p \nrightarrow 1$, let R be an arbitrary reduction of the string to sentence type. Then, by 3.7, R links each critical type t_q on the left with some t_{k_q}, such that $i < k_q < p$. Moreover, since the fan is guarded, the backward stack will become empty when the type $t_{k_{p+n}}$ is read. At this point, the segment $T^l := t_{j_p}....t_{j_{p+n}}$ is added to the forward stack. The remaining reductions are linear and T^l will be linearly reduced by Lazy parsing, since the fan is guarded. Thus, $MinPP$ will correctly reduce this string to the sentence type.

Inductive Hypothesis

Assume $MinPP$ parses any sentence with at most m guarded critical fans.

Inductive Step

Consider a string with $m+1$ guarded critical types. Consider the leftmost critical fan, and write $T^r := t_p...t_{p+n}$ for the segment given by its right end points. Let R be a reduction of the string to the sentence type. We have again two cases:

 # 1: Let R reduce T^r with T in the top of the stack computed by Lazy Parsing. $MinPP$ will reduce $TT^r \to 1$ by lazy parsing. After this stage, consider the string P obtained by appending the remaining unprocessed string to St. P contains m critical fans and reduces to sentence type, thus, by inductive hypothesis, $MinPP$ will parse it.

 # 2: Assume T^r does not reduce with types in the stack. Let $T := t^l_{p+n}...t^l_p$ be the types in the string which are reduced with T^r. Their index must be larger than i. Write $\theta := t^l_{p+n}...T^r$. Write θ_{prec} for the segment preceding θ, and θ_{post} for the segment following θ. θ_{prec} is linear, so its irreducible form D is unique. Moreover, by construction, we must have $D = CT^l$. Then $MinPP : \theta_{prec} \implies CT^l$ by Lazy Parsing. Since T^r is guarded, the backward stack will eventually be empty and $MinPP : \theta_{prec}\theta \implies CT^l$. The remaining string $CT^l\theta_{post}$ contains m guarded critical types and, since T^r is guarded, it has a reduction to sentence type. By inductive hypothesis, $MinPP : C\theta_{post} \implies s$.

□

Note that this proves that $MinPP$ is **correct** for the class of complexity 2 pregroup grammars identified by the above restrictions on the critical fans. We recall that complexity 2 grammars hold the same expressive power of the whole class of pregroup grammars. We now verify that $MinPP$ parses string in quadratic computational time.

Lemma 4.3. $MinPP$ parses a string in time proportional to the square of the length of the string.

Proof. Let N be the number of simple types in the processed string. $MinPP$ sees each type exactly once in forward processing. This includes either attempting reductions with the top of the stack or searching for a critical fan. In both cases these processes are obtained via functions with constant time d. Thus the forward processing happens overall in time dN. Then, for each critical fan, we read the string backward. This process is done in time dN^2 at most. Finally, when backward critical reductions are found, we correct the stack and set of reductions. The correction functions have constant time c, so all corrections happen in time cN at most. Summing these terms we obtain:

$$time = dN^2 + (d + c)N.$$

□

5 $LinPP$: Linear Pregroup Parsing algorithm

Certain words are typically assigned compound types by the dictionary, e.g. $T := n^r s n^l$ for *transitive verbs*. It might be the case that a compound type T_W of a word W, is not irreducible. Both $MinPP$ and the parsers mentioned in the Introduction will read types in T_W and reduce T_W to an irreducible form. However, the main purpose of grammatical pregroup types is to tell us how to connect different words. Reducing words internally defeats this purpose. We want to overcome this limitation and construct an algorithm that ignores intra-word reductions. Given a word W_1 let T_1 be its corresponding type (simple or compound). In $MinPP$ we defined stages S_n corresponding to each simple type t_n being read. Let us write Z_1 for the *super stage* corresponding to word W_1 being read. Z_1 contains one or more S_n corresponding to each simple type in T_1. We modify $MinPP$ as follows.

- At stage Z_1, we add $T_1 = t_1...t_j$ to the stack. We immediately jump to super stage Z_2 and stage S_{j+1}.
- When each new word W_m, with $m > 1$ and $T_m := t_{m_1}...t_{m_k}$ is processed, We try to contract $t_i t_{m_1}$. While types contracts we keep reducing the types t_{m_j} with the top of the stack. We stop when either a pair $t_i t_{m_j}$ does not contract or when we reach the end of the word.
- If $t_i t_{m_j} \nrightarrow 1$ and t_{m_j} is linear, we add $t_{m_j}...t_{m_k}$ to the stack and jump to stages Z_{m+1}, S_{m_k+1}. If t_{m_j} is critical, we handle it as in $MinPP$: if a backward reduction is found, the stack and reduction set are updated and we move to S_{m_j+1}; if the backward reduction is not found, we add $t_{m_j}...t_{m_k}$ to the stack and move to the next word as above.

In other words, $LinPP$ follows the same computational steps of $MinPP$, while only checking reductions between types of separate words. By assuming dictionaries whose sentences do not involve intra-word reductions, the above proof of correctness can be adapted to hold for $LinPP$. Modifications are trivial. We previously highlighted the importance of a linear parser; up to this point $LinPP$ computes parsing in quadratic time. Below we impose some further restrictions on the input data, which enable linear computational time.

Definition 5.1. We say that a dictionary of complexity 2 is **critically bounded** if, given a constant $K \in \mathbb{N}$, for each critical type t_c in a string, exactly one of the following is true:

- t_c reduces when processing the substring $t_{c-K}...t_c$ backwards;
- t_c does not not reduce in the string.

In other words, critical underlinks cannot exceed lenght K.

Lemma 5.2. Assume the restrictions of section 4.1, no-intra word reductions, and critically bounded dictionaries. Then $LinPP$ parses strings in linear computational time.

Proof. Assume a string of length N. $LinPP$ forward processing involves reading each type at most once. Thus it happens at most in time dN, with d as in section 4.1. Moreover, when a critical fan is read, the string is parsed backward, reading at most K types. This process takes dK time per critical fan. Thus it takes overall times dKN. Finally there is an extra linear term, cN, given by the time spent to correct the stack and reduction set. Summing up those terms, we obtain overall computational time CN, with $C = d(1+K)+c$ being a constant specific to each bounded dictionary. □

6 Conclusion

In this paper we first defined a quadratic pregroup parser, $MinPP$, inspired by Preller's minimal parser. We proved its correctness with respect to reducing strings to irreducible forms, and in particular to parse sentences to the sentence type, in the class of pregroup grammar charactorised by complexity 2 and guarded critical types. Note that our definition of guards differs from the one given in (Preller, 2007a). We then modified $MinPP$ in order to remove intra-words links. We proved that the obtained algorithm, $LinPP$, is linear, given that the dictionaries are critically bounded. $LinPP$ was implemented in Python and it's soon to be integrated in the *DisCopy* package. The reader can find it at github:*oxford-quantum-group/discopy*. $LinPP$ is an important step towards the implementation of a supervised pregroup tagger, which will enable extensive testing of the DisCoCat model on

task involving larger data-sets. Future theoretical work and implementations will involve researching a probabilistic pregroup parser based on $LinPP$. Future work might also involve investigation the connection between pregroup parsers and compositional dynamical networks.

Acknowledgments

The author thanks Giovanni Defelice and Alexis Toumi for the constructive discussions and feedback on the parser and its Python implementation. The author thanks their supervisors, Bob Coecke and Stefano Gogioso, for directions and feedback. Many Thanks to Antonin Delpeuch for insights on cubic pregroup parsers. Last but not least, the author thanks Anne Preller for the precious input in reformulating the definition of *guards* and for the insightful conversation on the topic of this paper.

References

W. Buszkowski. 2009. Lambek grammars based on pregroups. *Logical Aspects of Computa- tional Linguistics, LNAI 2099*.

B. Coecke. 2019. The mathematics of text structure. *arXiv: 1904.03478 [cs.CL]*.

B. Coecke, M. Sadrzadeh, and S. Clark. 2010. Mathematical foundations for a compositional distributional model of meaning. *arXiv:1003.4394v1 [cs.CL]*, pages 1–34.

Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, and Alexis Toumi. 2020a. Foundations for Near-Term Quantum Natural Language Processing. *arXiv:2012.03755 [quant-ph]*.

Bob Coecke, Giovanni de Felice, Konstantinos Meichanetzidis, Alexis Toumi, Stefano Gogioso, and Nicolo Chiappori. 2020b. Quantum natural language processing.

G. Defelice, A. Toumi, and B. Coecke. 2020. Discopy: Monoidal categories in python. *arXiv:2005.02975 [math.CT]*.

S. Degeilh and A. Preller. 2005. Efficiency of pregroups and the french noun phrase. *Journal of Language, Logic and Information*, 4:423–444.

J. Earley. 1970. An efficient context-free parsing algorithm. *Communications of the AMC*, 13:94–102.

J. Lambek. 1997. Type grammars revisited. *LACL 1997*, pages 1–27.

Konstantinos Meichanetzidis, Stefano Gogioso, Giovanni De Felice, Nicolò Chiappori, Alexis Toumi, and Bob Coecke. 2020a. Quantum Natural Language Processing on Near-Term Quantum Computers. *arXiv:2005.04147 [quant-ph]*.

Konstantinos Meichanetzidis, Alexis Toumi, Giovanni de Felice, and Bob Coecke. 2020b. Grammar-Aware Question-Answering on Quantum Computers. *arXiv:2012.03756 [quant-ph]*.

K. Moroz. 2009a. Parsing pregroup grammars in polynomial time. *International Multiconference on Computer Science and Information Technology*.

K. Moroz. 2009b. A savateev-style parsing algorithm for pregroup grammars. *International Conference on Formal Grammar*, pages 133–149.

A. Preller. 2007a. Linear processing with pregoups. *Studia Logica*.

A. Preller. 2007b. Towards discourse prepresentation via pregroup grammars. *JoLLI*, 16:173–194.

D. Shiebler, A. Toumi, and M. Sadrzadeh. 2020. Incremental monoidal grammars. *arXiv:2001.02296v2 [cs.FL]*.

William Zeng and Bob Coecke. 2016. Quantum Algorithms for Compositional Natural Language Processing. *Electronic Proceedings in Theoretical Computer Science*, 221:67–75.