Efficiency of Pregroups and the French Noun Phrase

Sylvain Degeilh, Anne Preller

Sylvain Degeilh
LIRMM/CNRS
Montpellier
France
degeilh@lirmm.fr

Anne Preller
LIRMM/CNRS
Montpellier
France
preller@lirmm.fr

Efficiency of pregroups and the French noun phrase

ABSTRACT:

We study mathematical and algorithmic properties of Lambek's pregroups and illustrate them by the French noun phrase. An algorithm of complexity n^3 to solve the reduction problem in an arbitrary free pregroup as well as recognition by a pregroup grammar is presented. This algorithm is then specified to run in linear time. A sufficient condition for a language fragment that makes the linear algorithm complete is given.

Introduction

Pregroups are a recent mathematical tool introduced in [Lambek 99] for natural language processing. They arose as a simplification of the Lambek calculus, first published under the name of Syntactic Calculus in [Lambek 58]. A pregroup grammar consists of a free pregroup and a pregroup dictionary, i.e. a finite relation which associates finitely many elements of the pregroup, the so-called *types*, to each word. [Buszkowski] has shown that pregroup grammars are weakly equivalent to context-free grammars. Well known algorithms like [CYK] or [Earley] for context-free grammars solve the problem of the grammaticality of strings of words in time proportional to the cube of the length of the string. However, it would not only be clumsy and inefficient to translate a pregroup grammar to a context-free grammar (this grammar would include the whole dictionary in its set of rules), but it would also go against the pregroups spirit. Indeed, pregroups were conceived as a tool which "processes the words as you hear them". For each word you hear, you choose a type from the (mental) dictionary. The grammaticality of a string of words from the dictionary is then decided by performing computations on the corresponding string of types in the pregroup. So far, much of the work with pregroups has been aimed at establishing the universality of the rules, which are the same whatever the language - only the dictionary changes. Fragments of Arabic, Japanese, Latin [Math-Ling], Polish [Kislak], English [Lambek 99, 03], French [Barg Lamb], German [Lamb-Prel NP, SE], Italian [Cas-Lamb] among others have been analysed with the help of pregroups.

This paper presents an algorithm which solves the decision problem of the theory of pregroups as well as recognition by a pregroup grammar in time proportional to the cube of the length n of the input string.[1] As this algorithm combines type assignment and type checking, the constant of n^3 depends on a bound for the number of types per word and a bound for their length, but not on the number of rules or symbols, which may even be infinite. This is still an improvement on known algorithms for context-free grammars, where the constant depends on the size of the dictionary. Since "type checking" is a special case of the recognition algorithm, the former is then simplified to run in *linear* time. This linear algorithm is generally not complete, so a simple condition on the set of input types is given, making the linear algorithm complete. This linearity condition is not satisfied by all dictionaries, but its violation is linked to certain grammatical constructs, e.g. post-nominal adjectives of a French noun phrase.

In the second part of the paper, representing the starting point of this work, we extend the dictionary given in [Barg-Lamb] to include agreement and structure of the noun phrase. This is achieved by adding several new basic types, organised according to features. Our extension is *conservative* in the sense defined in Section 1, i.e. the new constructs and new words can be added without changing the previous analysis conducted for the smaller grammar. A proliferation of basic types does not affect efficiency, but a proliferation of types per word does. Unification of features can be used to increase the type assignment efficiency, without effecting the pregroup grammar.

The small part of the French noun phrase covered here illustrates how the linearity condition is violated. In English or German, relative clauses specifying noun phrases would have the same effect. It also shows that features can be used in pregroup grammars, but only as a device to balance the increase in complexity due to an increased number of types per word. The dictionary presented here is the starting point to ongoing work ([Degeilh], [Preller]) on the efficiency and expressive power of pregroup grammars.

Finally, we would like to thank two anonymous referees for their constructive criticism. We have tried to address most of their concerns.

1. Mathematical properties

We briefly review the definitions.

<u>Definition 1</u>: A pregroup is a partially ordered monoid in which each element a has both a *left adjoint* a^ℓ and a *right adjoint* a^r satisfying

---

[1] After submitting this paper, we learned that [Oehrle] independently found a cubic time algorithm tailored to pregroups.

(Contraction) $\quad a^{\ell} \cdot a \rightarrow 1$

$\qquad\qquad\quad a \cdot a^{r} \rightarrow 1$

(Expansion) $\quad 1 \rightarrow a^{r} \cdot a$

$\qquad\qquad\quad 1 \rightarrow a \cdot a^{\ell}\,.$

The following are immediate consequences:

Properties:

1. $a1 = a = 1a$ — (1 is the unit of the monoid)

2. $(ab)c = a(bc)$ — (multiplication is associative)

3. $a \rightarrow b$ and $c \rightarrow d$ implies $ac \rightarrow bd$ — (order is compatible with multiplication).

4. $ab \rightarrow 1 \rightarrow ba$ implies $a = b^{\ell}$ and $b = a^{r}$ — (adjoints are unique)

5. $(ab)^{\ell} = b^{\ell}a^{\ell}$, $(ab)^{r} = b^{r}a^{r}$ — (adjunction is quasi-distributive)

6. $a \rightarrow b$ implies $b^{\ell} \rightarrow a^{\ell}$ and $b^{r} \rightarrow a^{r}$ — (adjunction reverses the order).

7. $a^{\ell r} = a = a^{r\ell}$ — (no mixed adjoints)

8. $a \rightarrow b$ if and only if $a^{\ell}b \rightarrow 1$ if and only if $ab^{r} \rightarrow 1$

Properties 1. – 3. are part of the definition of a partially ordered monoid, while the Properties 4. - 7. can be easily derived from Definition 1. For example, to derive that $b = b^{\ell r}$ (Property7), use $b^{\ell}b \rightarrow 1 \rightarrow bb^{\ell}$ and 4. with $a = b^{\ell}$. Similarly, 5. follows from 4. Indeed, $(ab)(b^{r}a^{r}) = a(bb^{r})a^{r} \rightarrow a1a^{r} = aa^{r} \rightarrow 1 \rightarrow b^{r}b = b^{r}1b \rightarrow b^{r}(a^{r}a)b = (b^{r}a^{r})(ab)$. Hence, by 4., $(ab)^{r} = b^{r}a^{r}$. Finally Property 8. is shown as follows: If $a \rightarrow b$, then $b^{\ell}a \rightarrow b^{\ell}b \rightarrow 1$ by 3. and Contraction. If $b^{\ell}a \rightarrow 1$, then $a \rightarrow bb^{\ell}a \rightarrow b$, using 3. and Expansion. [2]

Important additional properties apply in free pregroups. First of all, they are non-commutative and the iterated adjoints of basic types are all different. The most important result is expressed in the Switching Lemma [Lambek 99, Proposition 2], which we shall include here for completeness sake.

The *free* pregroup generated by a partially ordered set of *basic types*

$$B = \{a, b, ...\}\,.$$

is characterised in [Lambek 99] as the ordered free monoid generated from the set of *simple types* Σ consisting of the basic types and their iterated adjoints:

---

[2] We are grateful to Joachim Lambek for having drawn our attention to this.

$$\Sigma = \{ ...\boldsymbol{a}^{(-2)}, \boldsymbol{a}^{(-1)}, \boldsymbol{a}^{(0)}, \boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, ... \qquad , ...\boldsymbol{b}^{(-2)}, \boldsymbol{b}^{(-1)}, \boldsymbol{b}^{(0)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, ... \}$$

where

$$...\boldsymbol{a}^{(-2)}, \boldsymbol{a}^{(-1)}, \boldsymbol{a}^{(0)}, \boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, ...$$

stands for

$$...\boldsymbol{a}^{\ell\ell}, \boldsymbol{a}^{\ell}, \boldsymbol{a}, \boldsymbol{a}^{r}, \boldsymbol{a}^{rr}, ...$$

In particular, $\boldsymbol{a}^{(0)}$ stands for $\boldsymbol{a}$. Unit 1 denotes the empty string (thus is not an element of Σ), multiplication is the same as concatenation. Let n be an integer. Declare $\boldsymbol{a}^{(n)} \to \boldsymbol{b}^{(n)}$, if either n is even and $\boldsymbol{a} \to \boldsymbol{b}$ or n is odd and $\boldsymbol{b} \to \boldsymbol{a}$. By definition, every type that differs from 1 has the form

$$\boldsymbol{a}_1^{(n_1)}...\boldsymbol{a}_k^{(n_k)}$$

where $\boldsymbol{a}_1,...,\boldsymbol{a}_k$ are basic types and $n_1,...,n_k$ are integers. The adjoints of a type are defined by

$$(\boldsymbol{a}_1^{(n_1)}...\boldsymbol{a}_k^{(n_k)})^{\ell} := \boldsymbol{a}_k^{(n_k-1)}...\boldsymbol{a}_1^{(n_1-1)}$$

$$(\boldsymbol{a}_1^{(n_1)}...\boldsymbol{a}_k^{(n_k)})^{r} := \boldsymbol{a}_k^{(n_k+1)}...\boldsymbol{a}_1^{(n_1+1)}$$

The order on types $a \to b$ is now read as "a reduces to b". It is defined as the transitive closure of the union of the following three relations

(Induced step)

$$c\boldsymbol{a}^{(n)}d \to c\boldsymbol{b}^{(n)}d \text{, if either } n \text{ is even and } \boldsymbol{a} \to \boldsymbol{b} \text{ or } n \text{ is odd and } \boldsymbol{b} \to \boldsymbol{a}.$$

(Generalised contraction)

$$c\boldsymbol{a}^{(n)}\boldsymbol{b}^{(n+1)}d \to cd \text{, if either } n \text{ is even and } \boldsymbol{a} \to \boldsymbol{b} \text{ or } n \text{ is odd and } \boldsymbol{b} \to \boldsymbol{a}.$$

(Generalised expansion)

$$cd \to c\boldsymbol{a}^{(n+1)}\boldsymbol{b}^{(n)}d \text{, if either } n \text{ is even and } \boldsymbol{a} \to \boldsymbol{b} \text{ or } n \text{ is odd and } \boldsymbol{b} \to \boldsymbol{a}.$$

where c,d are arbitrary types and $\boldsymbol{a},\boldsymbol{b}$ are basic.

In the following, a *substring* of $a_1...a_n$ always means a substring $a_{i_1}...a_{i_m}$ in which the symbols appear in the same order as in the original string: $1 \le i_1 < ... < i_m \le n$.

<u>Switching Lemma [Lambek 99]</u>: Let $a_1,...,a_n$ and $b_1,...,b_m$ be simple types. Then $a_1...a_n \to b_1...b_m$ if and only if there is a substring $a_{i_1}...a_{i_k}$ of $a_1...a_n$ and a substring $b_{i_1}...b_{i_k}$ of $b_1...b_m$ such that

$$a_1...a_n \to a_{i_1}...a_{i_k} \to b_{i_1}...b_{i_k} \to b_1...b_m, \ a_{i_p} \to b_{i_p}, 1 \le p \le k ,$$

where $a_{i_1}...a_{i_k}$ is obtained from $a_1...a_n$ by generalised contractions only, $b_1...b_m$ is obtained from $b_{i_1}...b_{i_k}$ by generalised expansions only and $b_{i_1}...b_{i_k}$ is obtained from $a_{i_1}...a_{i_k}$ by induced steps only.

<u>Corollary [Lambek 99]</u>: Suppose that $a_1,...,a_n$ and b are simple types. Then $a_1...a_n \to b$ holds if and only if there is a simple type $b' \to b$ such that one can obtain b' from $a_1...a_n$ by repeatedly omitting

pairs of contractible adjacent types, i.e. there is a sequence $X_1 = a_1 ... a_n$, $X_2, ..., X_m = b'$ such that $X_i \rightarrow X_{i+1}$ is a generalised contraction for $1 \leq i < m$. Similarly, $a_1 ... a_n \rightarrow 1$ if and only if the empty string can be obtained by repeatedly omitting pairs of contractible adjacent types.

An immediate consequence of the Switching Lemma is the decidability of the theory of pregroups, i.e. the problem whether $a \rightarrow b$ is decidable for arbitrary types a and b in an arbitrary free pregroup. We call this the *reduction problem for free pregroups*. Any decision procedure of this problem is called a *type checking* algorithm. By Property 8, Section 1, any decision procedure of the particular problem $a \rightarrow 1$ also yields a decision procedure for the general problem.

Finally, here some more consequences of the Switching Lemma which will be useful in the following sections:

Proposition 1: (Conservativity of extensions) : Let B be an ordered subset of B', i.e. B is a subset of B' and $a \rightarrow b$ in B if and only if $a, b \in B$ and $a \rightarrow b$ in B'. Then the free pregroup P' generated by B' is conservative over the free pregroup P generated by B. That is to say, for all elements e, f of P such that $e \rightarrow f$ holds in P', $e \rightarrow f$ holds already in P.

Proof: Let e, f be elements of P such that $e \rightarrow f$ holds in P'. First consider the special case where $e \rightarrow f$ is a generalised contraction. Then there are basic types $a, b \in B$ and an integer n such that $e = c a^{(n)} b^{(n+1)} d$, $f = cd$ and either $a \rightarrow b$ in B' and n is even or $b \rightarrow a$ in B' and n is odd. By hypothesis, this implies $a \rightarrow b$ in B and n even, or $b \rightarrow a$ in B and n odd. Therefore $e \rightarrow f$ is a generalised contraction in P. By a similar argument, a generalised expansion (respectively induced step) in P' also is a generalised expansion (respectively induced step) in P.

In the general case, there are simple types $a_1, ..., a_n$ and $b_1, ..., b_m$ of P such that $e = a_1 ... a_n$, $f = b_1 ... b_m$ and $a_1 ... a_n \rightarrow b_1 ... b_m$ in P'. Apply the Switching Lemma. There is a substring $a_{i_1}, ..., a_{i_k}$ of $a_1, ..., a_n$ and a substring $b_{i_1}, ..., b_{i_k}$ of $b_1, ..., b_m$ such that

$$a_1 ... a_n \rightarrow a_{i_1} ... a_{i_k} \rightarrow b_{i_1} ... b_{i_k} \rightarrow b_1 ... b_m, \quad a_{i_p} \rightarrow b_{i_p} \text{ in } P', \quad 1 \leq p \leq k,$$

where $a_{i_1} ... a_{i_k}$ is obtained from $a_1 ... a_n$ by repeated generalised contractions and $b_1 ... b_m$ is obtained from $b_{i_1} ... b_{i_k}$ by repeated generalised expansions. Now for every generalised contraction $c \rightarrow d$, from $c \in P$ follows $d \in P$. Indeed, d is obtained from c by omitting two simple types. Similarly, for every generalised expansion $c \rightarrow d$, from $d \in P$ follows $c \in P$. Hence $a_{i_1} ... a_{i_k}$ and $b_{i_1} ... b_{i_k}$ are in P and all the generalised contractions establishing $a_1 ... a_n \rightarrow a_{i_1} ... a_{i_k}$, induced steps used

for $a_{i_1}...a_{i_k} \to b_{i_1}...b_{i_k}$ and generalised expansions intervening in the derivation of $b_{i_1}...b_{i_k} \to b_1...b_m$, take place in P. By the first part of the proof, this implies that $a_1...a_n \to a_{i_1}...a_{i_k} \to b_{i_1}...b_{i_k} \to b_1...b_m$, and $a_{i_p} \to b_{i_p}$, $1 \le p \le k$, already hold in P.

Starting from the same string, different choices may lead to different results. For example, $a^\ell aa^r \to a^\ell$ and $a^\ell aa^r \to a^r$, indeed, $a^\ell aa^r = a^\ell(aa^r) \to a^\ell$ (contract aa^r) and $a^\ell aa^r = (a^\ell a)a^r \to a^r$ (contract $a^\ell a$). Generally, parentheses are a useful device for suggesting the generalised contractions justifying a reduction: $a^{\ell\ell}a^\ell aa^r = (a^{\ell\ell}a^\ell)(aa^r) \to 1$ (contract the left most and the right most pair) and $a^{\ell\ell}a^\ell aa^r = a^{\ell\ell}(a^\ell a)a^r \to a^{\ell\ell}a^r$ (contract the central pair). Both times we end up with a string where we can "go no further". Hence,

<u>Definition 2</u>: A pair of simple types (c,d) is *contractible*, if $c = a^{(n)}, d = b^{(n+1)}$ and either $a \to b$ and n even or $b \to a$ and n odd. A string of simple types is *irreducible*, if it has no adjacent simple types which are contractible. An *irreducible form* of $a_1...a_n$ is a substring that is irreducible. A string of three simple types abc is called a *critical triple*, if $ab \to 1$ and $bc \to 1$. A string of simple types $a_1...a_n$ is *linear*, if no substring $a_{i_1}a_{i_2}a_{i_3}$ is a critical triple.

For example, $a^r aa^\ell$ is linear, but $a^\ell aa^r$ is not linear. Or if $a \nrightarrow b$, then $cb^\ell aa^r d$ is linear, but if $a \to b$, then $cb^\ell aa^r d$ is not linear. Linearity is a sufficient condition for a type to have a unique irreducible form. For such types, every algorithm which produces some irreducible form will have found all irreducible forms and thus will be a type checking algorithm. Obviously, the amount of work needed to find one irreducible form is much lower than to find them all.

<u>Proposition 2</u>: Every linear type has a unique irreducible form.
Proof: Suppose $a_1...a_n$ is linear. Use induction on n, the length of the string. If $n = 1$, the property is obvious. For the induction step, note that every substring of $a_1...a_n$ is again linear. Moreover, whenever $a_i a_{i+1}$ and $a_j a_{j+1}$ are different contractible pairs, indices $i, i+1, j, j+1$ are all different. Suppose $a_1...a_n$ has k pairs of contractible adjacent types. Omitting them in $a_1...a_n$ corresponds to k simultaneous contractions. They can be done in any order without changing the result. If $k = 0$, then $a_1...a_n$ is irreducible. Otherwise, the unique substring resulting from the k contractions is less than n in length and we may conclude by the induction hypothesis.

Finally, an almost self-evident property, which we retain for later use.
<u>Lemma 1</u>: Suppose $a_1...a_n \to 1$. Then there is an index $j < n$ such that $a_j a_n \to 1$, $a_{j+1}...a_{n-1} \to 1$ and $a_1...a_{j-1} \to 1$.

Proof: Let $X_1 = a_1...a_n$, $X_2,...,$ $X_m = 1$ be a reduction of $a_1...a_n$ to the empty string obtained by generalised contractions only, i.e. $X_i \rightarrow X_{i+1}$ is a generalised contraction for $1 \le i < m$. Let i be the index such that a_n occurs in X_i but not in X_{i+1}. Then X_i and X_{i+1} have the form $X_i = Ua_ja_n$, $X_{i+1} = U$. Generalised contractions involving at least one simple type a_k strictly between a_j and a_n appear before X_i and therefore a_k is contracted with a type which is also strictly between a_j and a_n. Hence $a_{j+1}...a_{n-1} \rightarrow 1$. The argument for $a_1...a_{j-1} \rightarrow 1$ is similar.

2. Linguistic applications

A linguist will work with a free pregroup. The partially ordered set of basic types B generating the free pregroup is tailored to the language fragment under consideration. As a first step, the language fragment has to be described in common grammatical terms. We select words that are supposed to be in the mental or electronic dictionary, e.g. nouns, adjectives etc. Grammatical concepts describing grammatical constructs of the fragment will help to choose the basic types and the ordering. Finally, one or several types are associated with every word of the fragment, thus constituting a dictionary. Meta-rules serve to organise the content of the dictionary. Instead of explicitly writing the type(s) of a word in the dictionary, it (they) may be described by a meta-rule.

The dictionary must be designed in such a way that a sequence of words is a grammatical construct (sentence, noun phrase, etc) if and only if one of the corresponding strings of types reduces to a basic type. If a word has more than one type, then it is sufficient that one of the possible choices (*type assignments)* yields a string reducing to the basic type in question.

Every dictionary that respects this equivalence is said to be *correct* (it recognises only grammatical constructs) and *complete* (it recognises all grammatical constructs) of the fragment. However, correctness and completeness are generally not proven, but only illustrated on examples and counter-examples, unless the language fragment is already given by a formal grammar.

A correct and complete dictionary satisfies the *principle of substitution* and certain *robustness* properties:

Principle of substitution: If a word is replaced by another word with the same type, then a grammatical string of words remains grammatical. Indeed, being a grammatical string of words is equivalent to having a string of types reducible to a given basic type and, by assumption, the string of types is the same before and after substitution.

Robustness a): Assigning new types to words:

Suppose a word w has type d and we also give it type c such that $c \rightarrow d$, then every string of words that is accepted using d for w is also accepted using c.

Robustness b): Extensions by new basic types.

This means that one can extend a given set B of basic types, by declaring new types and adding inequalities involving the new types, thus obtaining a larger set of basic types B'. Then the free pregroup P' generated by B' includes the free pregroup P generated by B. Whenever both a and b belong to the smaller pregroup P and $a \to b$ can be derived in P, then this also holds in the larger P'. By Proposition 1 above, the converse also holds, provided that the order in B remains unchanged: if $a \to b$ can be derived in the larger pregroup P' and both a and b belong to the smaller pregroup, then the whole reduction can be done in the smaller pregroup.

The linguistic significance of this is that the language fragment can be increased by adding new basic types and/or new types to the dictionary, without having to repeat verification of correctness and completeness performed before the extension. It also simplifies the task of verifying that a typing is correct and complete. One can proceed step by step, extend the fragment by adding new basic types and assigning new types. Then to show that the typing is correct and complete with respect to the larger fragment, only the sequences of words with new types have to be considered. The only prerequisite is not to change the order between the old basic types. For example, suppose $B' = B[c]$ where c is a new basic type. We may declare $a \to c$ and/or $c \to b$ for some "old" basic types $a,b \in B$. However, we must take care not to declare both $a \to c$ and $c \to b$, unless $a \to b$ already holds in B. Note that this step by step approach is normally taken for granted. This is not as trivial as one might think. Obviously, a sequence of words that is a grammatical construct of the smaller fragment remains so in the larger fragment. Just use the typing and the reduction in the smaller pregroup P, which remains a reduction in the bigger one P'. It is, however, not so obvious that a sequence of words which is not grammatical in the smaller fragment does not become grammatical in the bigger fragment, even if no new types are involved. Indeed, suppose that a sequence of words gets assigned a type c in the smaller fragment, but is not grammatical. As the typing in P is correct and complete with respect to the smaller fragment, we have $c \nrightarrow a$ where a is the basic type corresponding to the grammatical notion under investigation. A priori, it could not be excluded that $c \to d$ and $d \to a$, where d is a new type in P'. But then, it would follow that $c \to a$ in P'. From this we would have to conclude that the sequence of words is now well-formed, because its type reduces to a. Conservativity ensures that this cannot happen, as from $c \to a$ in P' it would follow that $c \to a$ in P, contradicting $c \nrightarrow a$.

This property is actually used continuously and most of the time without saying so. We will do so in Section 4.

We will sum up the mathematically relevant facts of the discussion above:

<u>Definition 3</u>: 1) Let V be a non-empty set, B a partially ordered set and P the free pregroup generated by B. A *dictionary of vocabulary V with types in P*, is a map D from V to the set of subsets of P.

2) A dictionary D is *bounded* if there are constants k and l such that for every *word* $v \in V$ the set $D(v)$ has at most k elements, and each type in $D(v)$ has at most length l. A dictionary D is *locally finite*, if $D(v)$ is finite for all $v \in V$. It is said to be *finite*, if the sets V, B and $D(v)$ are finite.

3) A *type-assignment* for a string $v_1...v_n$ of elements in V is a sequence $t_1...t_n$ of types in P such that $t_i \in D(v_i)$, for $1 \le i \le n$.

4) Let a be a simple type in P. A string $v_1...v_n$ of elements in V is *a-grammatical*, if it has a type assignment $t_1...t_n$ such that $t_1...t_n \to a$. A sequence $v_1...v_n$ is *grammatical* if it is *a-grammatical* for some simple type a. Finally, $v_1...v_n$ is *well-formed* if there is a type assignment $t_1...t_n$ such that $t_1...t_n \to 1$.

3. Algorithmic Properties

Dictionaries defined for natural language fragments are generally assumed to be finite, i.e. sets V, B and $D(v)$ are finite. Such a finite dictionary together with a finite number of generalised contractions and induced steps involving types from the dictionary has been called a *pregroup grammar* in [Buszkowski]. It is shown in [loc.cit.] that pregroup grammars are weakly equivalent to context-free grammars. However, the context-free grammar associated with a pregroup grammar would include the whole dictionary in its set of rules. The complexity estimates of the algorithms in [CYK] or [Earley] use a constant factor for n^3 which must bound the number of symbols and rules of the grammar. However, there is no need to restrict oneself to finite dictionaries, the algorithm defined below works for an infinite number of rules and symbols.

In the following, we assume that D is locally finite, i.e. the sets $D(v)$ are finite, but not necessarily V or B. For example, suppose that V is equal to Σ, the set of simple types generated by B, and D equal to the identity map on Σ. Note that this dictionary is infinite, but bounded with bounds $k = 1 = l$. Moreover, the problem of grammaticality for this dictionary is the same as the decision problem for free pregroups, i.e. a recognition algorithm for this dictionary is also a type checking algorithm.

On the other hand, a type checking algorithm provides a solution to the problem of grammaticality for every dictionary D in which the sets $D(v)$ are finite. Indeed, try the type checking algorithm on all possible type assignments of the string $v_1...v_n$. An algorithm which provides $v_1...v_n$ with associated strings of types from the dictionary is called a *type assignment algorithm*. To enumerate all type

assignments would not be very efficient: If k_i is the number of elements in $D(v_i)$, then there are $k_1 k_2 ... k_n$ different type assignments for $v_1 ... v_n$. The recognition algorithm given below combines type assignment and type checking, it works for all dictionaries, independent of the pregroup, grammar or language.

Note that an algorithm which solves the problem of well-formed strings for arbitrary dictionaries implies a decision procedure for the problem of a-grammaticality for arbitrary simple types a. Indeed, add to the vocabulary V a new element $\bar{v}$ and extend D to $\bar{v}$ by letting $D(\bar{v}) = \left\{ a^r \right\}$. Then a sequence $v_1 ... v_n$ from V is a-grammatical if and only if $v_1 ... v_n \bar{v}$ is well-formed. This follows from Property 8 of Section 1 and the fact that the only possible type assignment for $\bar{v}$ is a^r.

The intuitive idea underlying the algorithm is as follows: We process the string of symbols $W = v_1 ... v_n$ from left to right, proceeding by *stages*. At each stage, we choose a symbol v_i represented by its index i, a type t in $D(v_i)$ and a position p in t. We examine the simple type(s) placed just to the left of this position in some type assignment and store it (them) in the memory, where they are kept as a left "parenthesis" awaiting contraction with a simple type that might come later. Moreover, each of them could also be a "right parenthesis" to some earlier simple type, a "left parenthesis" ready for contraction. In this case, the two types are contracted, which means that the "left parentheses" awaiting contraction at the earlier stage become available again, i.e. are stored in the memory at the present stage. Only the "nearest left parentheses" available for contraction has to be remembered. This defines a function Nlp_{DW} on the set of stages, which takes subsets of S_{DW} as values. Context permitting, we omit the subscripts.

We first look at an example, in the case where the vocabulary is the set of simple types and the dictionary is the identity map: Let $W = cb^{\ell} a a^r d$ where we assume $a \to b,\ d \to b$

Stage s	Predecessor s'	Type read $a_{s'}$	Test	$Nlp(s)$	Corresponding choice(s) of contractions
1	0	none	none	$\{0\}$	initialising
2	1	c	none	$\{1\}$	$(c$
3	2	b^ℓ	$cb^\ell \to 1$, no	$\{2\}$	$(c(b^\ell$
4	3	a	$b^\ell a \to 1$, yes	$\begin{cases} 3 \\ 1 \end{cases}$	$(c(b^\ell(a,$ $(c(b^\ell a)$
5	4	a^r	$aa^r \to 1$, yes $ca^r \to 1$, no	$\begin{cases} 4 \\ 2 \end{cases}$	$(c(b^\ell(a(a^r,\ (c(b^\ell a)(a^r,$ $(c(b^\ell(aa^r)$
6	5	d	$a^r d \to 1$, no $b^\ell d \to 1$, yes	$\begin{cases} 5 \\ 1 \end{cases}$	$(c(b^\ell(a(a^r(d,\ (c(b^\ell a)(a^r(d,\ (c(b^\ell(aa^r)(d$ $(c(b^\ell(aa^r)d)$

Note that this function does not remember the intermediary contractions already made. It only remembers the nearest not yet contracted type to the left.

Definition 4 (Nearest left parentheses function Nlp): Let D be a dictionary and $W = v_1...v_n$ a non-empty string of elements from the vocabulary. The stages associated to W form a set

$$S_{DW} = \{(i,t,p) : 1 \le i \le n,\, t \in D(v_i),\ 1 \le p \le length(t)\} \cup \{(0,0,0),(n+1,0,0)\}.$$

ordered alphabetically, i.e.

$$(i',t',p') \le (i,t,p) \text{ if and only if either } i' < i \text{ or else } i' = i,\, t' = t,\, p' \le p.$$

In general, this order is not total, a stage may have several predecessors or successors. For example, the set of predecessors of (i,t,p) satisfies

$$Predecessor(i,t,p) = \begin{cases} \{(i,t,p-1)\}, & \text{if } p > 1 \\ \{(i-1,t',length(t')) :\ t' \in D(v_{i-1})\}, & \text{else} \end{cases}$$

However, if $t' \in D(v_{i-1})$ is chosen, then (i,t,p) has a unique predecessor compatible with this type assignment, namely $(i,t,p-1)$ if $p > 1$ and $(i-1,t',length(t'))$ if $p = 1$. If the context makes the type assignment clear, we denote this predecessor by $pred(s)$, and similarly for successors.

Finally, let a_s denote the simple type occupying position p in the type t of $s = (i,t,p) \in S_{DW}$. The *Nearest left parentheses* function is defined on the set of states as follows:

$$Nlp_{DW}(0,0,0) = \varnothing$$

$$Nlp_{DW}(s) = \bigcup_{s' \in Predecessor(s)} \left(\{s'\} \cup \bigcup_{s'' \in Nlp(s'),\, a_{s''}a_{s'} \to 1,\, s'' \ne (0,0,0)} Nlp(s'') \right).$$

When the vocabulary is the set of simple types and the dictionary is the identity map, a state has the form $(i, a_i, 1)$. As the string $W = a_1 ... a_n$ is given state $(i, a_i, 1)$ can be identified with the integer i. The definition above simplifies to

$$Nlp_{DW}(0) = \varnothing$$

$$Nlp_{DW}(i+1) = \{i\} \cup \bigcup_{j \in Nlp(i),\ a_j a_i \to 1,\ j \neq 0} Nlp(j).$$

Proposition 3: There is a recognition algorithm which decides for every locally finite dictionary D and every string of symbols in the vocabulary, whether it is grammatical. If the dictionary is bounded with bounds k and l, then the complexity is $o(n^3)$ where n is the length of the string. The constant factor of n^3 is equal to $k^3 l^3$.

Proof: The proof consists of showing that

1) $W = v_1 ... v_n$ is well-formed if and only if $(0,0,0) \in Nlp(n+1,0,0)$

2) $Nlp(n+1,0,0)$ can be calculated in at most $k^3 l^3\ n^3$ steps.

This follows from the next 5 lemmas.

Lemma 2 : $s^* \in Nlp(s)$ implies $s^* < s$. In particular, if $s = (i, t, p)$, then $Nlp(s)$ has at most ikl elements, where k bounds the number of types associated to v_j, $1 \leq j \leq n$, and l the length of the corresponding types.

This follows immediately from the definition.

Lemma 3 : Let $s = (i, t, p)$, $s^* = (i^*, t^*, p^*)$, $t = a_1 ... a_l$ and $t^* = a^*_1 ... a^*_{l^*}$. If $s^* \in Nlp(s)$, then there is a type assignment $t_{i^*} = t^*$, $t_{i^*+1}, ..., t_{i-1}, t_i = t$ for $v_{i^*} v_{i^*+1} ... v_{i-1} v_i$ such that

$$X = a^*_{succ(s^*)} ... a^*_{l^*}\ t_{i^*+1} ... t_{i-1} a_1 ... a_{pred(s)} \to 1,$$

i.e. a_{s^*} is "*a left parenthesis ready for contraction with a_s*".

Indeed, use induction on s. If $s = (0,0,0)$ there is nothing to prove. Assume $s^* \in Nlp(s)$. By definition, s^* is either itself a predecessor of s or there is a predecessor s' of s and an element $s'' \in Nlp(s')$ such that $s^* \in Nlp(s'')$ and $a_{s''} a_{s'} \to 1$. In the former case, $i^* + 1 > i - 1$, X is the empty string and we are done. Assume the latter. The induction hypothesis applies to s'' and s'. Hence there are type assignments $t_{i^*} = t^*$, $t_{i^*+1}, ..., t_{i''-1}, t_{i''} = t''$ and $t_{i''} = t''$, $t_{i''+1}, ..., t_{i'-1}, t_{i'} = t'$ such that

$$Y = a^*_{succ(s^*)} ... a^*_{l^*}\ t_{i^*+1} ... t_{i''-1} a''_1 ... a''_{pred(s'')} \to 1, \quad Z = a''_{succ(s'')} ... a''_{l''}\ t_{i''+1} ... t_{i'-1} a'_1 ... a'_{pred(s')} \to 1$$

and

$$a_{s''} a_{s'} \to 1.$$

Finally, let $t_i = t$. As s' is a predecessor of s either $i' = i-1$ or else $i' = i$ and $t' = t$. In both cases, s' is the predecessor of s compatible with the type assignment $t_{i*} = t*, t_{i*+1}, ..., t_{i-1}, t_i = t$. Therefore, $X = Ya_{s''}Za_{s'} \to 1$.

<u>Lemma 4</u>: Let $s = (i,t,p)$, $s* = (i*,t*,p*)$, $t = a_1...a_l$, $t* = a*_1...a*_{l*}$ and $t_{i*} = t*, t_{i*+1}, ..., t_{i-1}, t_i = t$ be a type assignment for $v_{i*}v_{i*+1}...v_{i-1}v_i$ such that $X = a*_{succ(s*)}...a*_{l*} t_{i*+1}...t_{i-1}a_1...a_{pred(s)} \to 1$. Then $s* \in Nlp(s)$.

Again proceed by induction on s. If $s*$ is a predecessor of s, then $s* \in Nlp(s)$ by definition. Otherwise, let s' be the predecessor of s compatible with the given type assignment. As the string X reduces to the empty string, its last simple type, $a_{s'}$, is contracted with some simple type occurring to before $a_{s'}$ in X, see Lemma 1 in Section 1. Hence, there is a stage $s'' = (i'', t'' = t_{i''}, p'')$ such that $a_{s''}a_{s'} \to 1$, $a''_{succ(s'')}...a''_{l''} t_{i''+1}...t_{i'-1}a'_1...a'_{pred(s')} \to 1$ and $a*_{succ(s*)}...a*_{l*} t_{i*+1}...t_{i''-1}a''_1...a''_{pred(s'')} \to 1$. By the induction hypothesis, this implies that $s'' \in Nlp(s')$ and $s* \in Nlp(s'')$. So, $s* \in Nlp(s)$ by definition.

<u>Lemma 5</u>: $(0,0,0) \in Nlp_{v_1...v_n}(n+1,0,0)$ if and only if there is some type assignment $t_1,...,t_n$ for $v_1...v_n$ such that $t_1...t_n \to 1$.

Apply Lemma 3' and Lemma 4' to $s* = (0,0,0)$, $s = (n+1,0,0)$.

<u>Lemma 6</u>: Let k bound the number of types and l the length of the types in $D(v_j), 1 \le j \le n$. Then $Nlp(i,t,p)$ can be calculated from the previous values in at most $i^2k^2l^2$ steps. If k,l are bounds for the dictionary, then $Nlp(n+1)$ is of complexity $o(n^3)$ where the constant factor of n^3 is k^3l^3.

Proof: There are at most ikl stages less or equal to s. For each stage s'' less than s, we compare it with the predecessor(s) s' of s. If it is a predecessor, we add it to $Nlp(s)$ and are done. Otherwise, we check if it belongs to the set $Nlp(s')$. If this is not the case, we go to the next predecessor. If s'' belongs to $Nlp(s')$, we check if $a_{s''}a_{s'} \to 1$. If this is not the case, we go to the next predecessor. If, on the contrary, $a_{s''}a_{s'} \to 1$, we add the elements of $Nlp(s'')$ to $Nlp(s)$ and are done. By Lemma 2, there are at most $ikl - k_i$ stages in $Nlp(s'')$, where k_i is the number of predecessors of s. On the whole, we have executed at most ikl steps for s'', counting comparisons and copying as one-step operations. As we do this for every s'' less than s, it takes at most $i^2k^2l^2$ operations to calculate $Nlp(s)$. Finally, to calculate $Nlp(n+1,0,0)$, we must calculate all sets $Nlp(s)$. Hence the whole number of steps can be bounded by $n^3k^3l^3$.

<u>Corollary</u>: If there is a bound K such that $Nlp_{v_1...v_n}(s) < K$ for all stages s and strings $v_1...v_n$, then the complexity of Nlp is proportional to n.

In the case of the free pregroup reduction problem, where $k = 1 = l$, the constant factor of n^3 is 1. In the case of dictionaries for a natural language fragment, the length of a type l assigned to a word generally does not exceed 4 or 5. The number k of types assigned to a word may go up to 12 or more. Though this constant is generally considerably smaller than the size of the dictionary, any algorithm of complexity $o(n^3)$ is unlikely to be used without a machine. Using the corollary, a linear strategy for type checking can be obtained by a slight modification of Nlp. The idea is to open a new left parenthesis, only if necessary, i.e. if the considered position cannot be an expected "right parenthesis". Call the corresponding function Llp, the "lazy left parentheses" function. Recall that stages are of the form $(i, a_i, 1)$, $0 \le i \le n+1$ and identify with the integers in $\{0, 1, ..., n+1\}$ as $k = 1 = l$.

<u>Definition 5 (Lazy left parenthesis function *Llp*)</u>:

$$Llp(0) = \varnothing$$

$$Llp(i+1) = \begin{cases} \bigcup_{j \in Link(i)} Llp(j), \text{ if } Link(i) \ne \varnothing \\ \{i\} \text{ else} \end{cases}$$

where $Link(i) = \left\{ j \in Llp(i): \ 1 \le j, \ a_j a_i \to 1 \right\}$.

It is clear that $Llp(i+1)$ is included in $Nlp(i+1)$ and has exactly one element. Moreover, Llp defines a reduction and $j \in Link(i)$ means that a_j is contracted with a_i in this reduction. More precisely, say that i and j are *linked*, if $j \in Link(i)$ or $i \in Link(j)$. Then

<u>Lemma 7</u>: 1) If $k \in Llp(i+1)$, then $a_{k+1}...a_i \to 1$ and for all m such that $k+1 \le m \le i$ there is a p with $k+1 \le p \le i$, linked to m.

    2) If $j \in Link(i)$, then $a_j...a_i \to 1$.

    3) $m \in Link(p)$ implies, for all $i > p$ and all $j \in Llp(i)$, that $j > p$ or $j < m$.

    4) Every index p is linked to one index m at most.

Proof: 1) If $k = i$, there is nothing to show. Assume $k \in Llp(i+1)$ and $k+1 \le m \le i$. As $k < i$, there is $j \in Link(i)$ such that $k \in Llp(j)$. As $Link(i) \subseteq Llp(i)$, the induction hypothesis applies to $i - j$ and to $j - k$. Note that if $m = j$ or $m = i$, then m is indeed linked to some p with $k+1 \le p \le i$. 2) is an immediate consequence of 1). 3) Suppose $m \in Link(p)$, $i > p$ and $j \in Llp(i)$. Use induction on $i - p$.

In the case of $i = p+1$, we have $j \in Llp(m)$, as $Link(p) \neq \varnothing$. Hence $j < m$. If $i > p+1$, then either $j = i-1 > p$ or $j \in Llp(i-1)$ and therefore, by the induction hypothesis, $j > p$ or $j < m$. To see 4), note first that an index p cannot be linked to two different smaller indices, as $Link(p)$ has at most one element. By 3) an index m cannot be linked to two larger ones, say p and i with $p < i$. And it also cannot be linked to a smaller and a larger one, as $m \in Link(p)$, $p \in Link(i)$ would also contradict property 3).

Finally, the next and last Lemma confirms that the unlinked simple types of the string form an irreducible substring.

<u>Lemma 8</u> : Let $Unlinked = \{i_1, ..., i_q\}$ be the set of unlinked indices in increasing order. Then the following holds
I) every index less than i_1 (respectively between i_ℓ and $i_{\ell+1}$, respectively larger than i_q) is linked to some index below i_1 (respectively between i_ℓ and $i_{\ell+1}$, respectively larger than i_q).
II) a_{i_ℓ} and $a_{i_{\ell+1}}$ are not contractible.
Proof: Assertion I) follows from Lemma 7, 1) and 2). To show II), assume $i = i_{\ell+1}$ and let $j \in Llp(i)$. By choice of i, $a_i \nleq a_j^{\ r}$, i.e. a_j and a_i are not contractible. Hence, it is sufficient to show that $j = i_\ell$. In view of Lemma 7, 1), we only must show that j is not linked to any index. If j were linked to a smaller index, this would contradict Lemma 7, 3). If j were linked to an index greater than i, this would imply that i is linked by Lemma 7, 1). Finally, j cannot be linked to a larger index which would be less than i, because of Lemma 7, 1) and 4).

Thus the lazy left parenthesis function finds an irreducible form of a string, but generally, strings have more than one irreducible form. Below we give a sufficient condition for types to have a single irreducible form.

<u>Definition 5</u>: The typing of a language fragment is said to be *linear* if all strings of types corresponding to strings of words in the dictionary are linear.

Proposition 4: The type checking problem of a linear fragment can be decided by a linear algorithm.

Proof: By Proposition 3, it is suuficient to find an algorithm which is linear in the length of the string and produces an irreducible form of the string. For this all we have to do is add a new step at stage $i+1$: erase j and i from $\{1,...,n\}$, whenever the test $j \in Link(i)$ succeeds. Together with the two steps to calculate $Llp(i+1)$, at most four operations are performed at stage $i+1$.

In the next section, the French noun phrase, when formed from determiners, prenominal adjectives and nouns only, is a linear fragment, hence is recognised by the "lazy" linear algorithm. Postnominal adjectives introduce critical triples, hence other linear strategies must be found.

4. French noun phrase

Agreement in gender and number was ignored in [Barg-Lamb]. Pregroup grammars handle those "features" by a proliferation of basic types. This does not increase the running time of the recognition algorithm, as we have seen in Section 3. For example, we introduce the basic type n_{gn} to denote a complete noun phrase, depending on its *gender g* and *number n.* We postulate $n_{gn} \to n$, where n is a type used in [loc.cit.]. By Proposition 1, the typing proposed below is a conservative extension of the typing in [Barg-Lamb].

The noun phrases analysed below are either names or a determiner followed by a noun[3]. Adjectives may occur between the determiner and the noun or follow the latter. The prenominal adjectives precede the noun, while the postnominal adjectives follow it. A noun with correctly declined and correctly placed adjectives forms an incomplete noun phrase. A determiner transforms an incomplete noun phrase into a complete noun phrase, which may be a subject or an object in a sentence. The notion of determiner follows [Le bon usage], it includes the indefinite article *un, une*, the definitive article, *le, la, l', les* and its contracted forms with *de* namely *du, des*, the possessive and demonstrative pronouns *son, sa, ses*, *ce(t), cette, ces,* etc. as well as the preposition *de* preceded by an adverb of degree like *beaucoup, peu,* or of negation like *pas, point,* etc.

4.1 Nouns and Adjectives

Nouns as well as adjectives vary in gender and number. Therefore the type of a noun is indexed by g, which stands for 1 = masculine or 2 = feminine, and by n, where $n = $ 1 means singular and $n = 2$ means plural. The gender of a noun is given in the dictionary. The plural is in most cases formed by appending the letter *s* to the singular form, but we will not discuss this here and assume that the plural form is given in the dictionary. Nouns are count nouns like *chat, pomme* or

---

[3] To keep the paper within reasonable limits, we ignore situations where the noun alone is a complete noun phrase: *Elle était institutrice, je vous en fais juge, nous venons en train* etc.

mass nouns like *eau, pain, vent,* etc. but also *courage, beauté* and so on. A count noun has type c_{0gn}, a mass noun has type m_{0g1}, for example

chats	:	c_{012}
pomme	:	c_{021}
vin	:	m_{011}
eau	:	m_{021}

Count nouns generally have both a singular and a plural. Many mass nouns have no plural, for example *riz* . If a mass noun has a plural, we treat the plural form as count noun. For example

vins	:	c_{012}
eaux	:	c_{022}

The first index in the types above serves to distinguish a bare noun from the incomplete noun phrases formed from a noun embellished by adjectives. Indeed, adjectives are divided into prenominal adjectives and postnominal adjectives [4], for example (*un*) *bon vin blanc,* versus (*un*) **blanc vin bon.* The great majority are postnominal, but most prenominal adjectives are very common, like *beau, petit, mauvais, jeune,* etc. More than one adjective may precede or follow a noun. They must respect a certain hierarchy, e.g. (*le*) *beau petit chat,* (*un*) *autre beau petit chat*, but **le petit beau chat*. The postnominal adjectives are also divided into classes which cannot be interchanged, compare *un vin blanc pétillant*, with **un vin pétillant blanc.* If two adjectives should occupy the same position, they must be linked by a copula [5]. We assume that the classification of the adjectives into prenominal and postnominal hierarchy classes is known and can be looked up in the dictionary. We use Arabic digits for the prenominal classes $C_1, C_2, ...$, Roman ones for the postnominal classes $C_I, C_{II}, ...$, i.e. we have classes C_h where $h \in \{1, 2, ...\} \cup \{I, II, ...\}$. The lower the number of its class, the closer the adjective will be to the noun.

We introduce basic types x_{hgn} where $h \in \{1, 2, ...\} \cup \{0\} \cup \{I, II, ...\}$ and x stands for c or m :

chat:		c_{011}
chat noir	:	c_{I11}
petit chat, petit chat noir:		c_{111}
beau (petit) chat:		c_{211}

Moreover, a maximal type x_{gn} is convenient in cases where the hierarchy does not matter:

---

[4] Some adjectives may belong to both classes, especially if classic French or regional variations are also to be covered.

[5] This can be done with the usual polymorphic typing of the copula. To keep the paper within reasonable limits, we ignore this case.

$$x_{hgn} \rightarrow x_{gn}, \text{ for all } h.$$

Determiners and certain adjectives vary in form according to whether the following word starts with a vowel[6] or not. For example, *l'arbre, cet arbre, bel arbre, mon eau*, but **le arbre, ce arbre, *beau arbre, *ma eau*.[7] Every type x will have a copy x' used for words or sequences of words of which the first letter is a vowel. For example, *arbre*, respectively *eau* has type c'_{011} respectively m'_{021}. Similarly, the masculine singular of a few adjectives like *beau, vieux* has a variant *bel, vieil* to be used if the next word starts with a vowel.[8] The typing of *beau*, *bel*, etc. is therefore

$$\text{beau, vieux, ... } : \ x_{211} x_{h11}^{\ell}, \text{ for } h = 1, 0, I, II, ...$$

$$\text{bel, vieil, ... } \quad : \ x_{211} x'_{h11}^{\ell}, \ h = 1, 0, I, II, ...$$

where $x = c$, $x = m$.

The feminine singular and plural forms of these special adjectives have no such variant, most other adjectives only have masculine singular. The following meta-rule applies to them and to the feminine singular and plural forms of the special adjectives:

Meta-Rule (Adjectives):

Let A be an adjective and A_{gn} its declined form of gender g and number n.

1) If A belongs to the prenominal hierarchy class C_i, $i = 1, 2, ...$, then

$$A_{gn} \quad : \ x_{ign} y_{hgn}^{\ell}, \ h = i - 1, ..., 0, I, II, ...,$$

where

either $x = c$ and $y = c, c'$ or $x = m$ and $y = m, m'$, if A starts with a consonant,

either $x = c'$, $y = c, c'$ or $x = m'$, $y = m, m'$, if A starts with a vowel.

2) If A belongs to the postnominal hierarchy class C_i, $i = I, II, ...$, then

$$A_{gn} \quad : \ x_{hgn}^{\ r} x_{ign}, \text{ where } x = c, c', m, m', \ 0 \leq h < i.$$

Examples:

$$\text{vin} \qquad\qquad\qquad : \ m_{011}$$

$$\text{blanc} \qquad\qquad\quad : \ c_{011}^{\ r} c_{I11}, \ m_{011}^{\ r} m_{I11}$$

$$\text{vin blanc}$$

---

[6] The initial silent *h* is assimilated to an initial vowel.

[7] This phenomenon is even more pervasive in the spoken language where the otherwise silent terminal consonant of a word is pronounced, if the following word starts with a vowel.

[8] In the spoken language, every adjective ending in a silent consonant has a variant form with an audible last letter if followed by a word starting with a vowel.

$$m_{011}\ m_{011}{}^{r}m_{I11} \qquad\qquad \rightarrow m_{I11}$$

amande : c_{021}

blanche : $c_{021}{}^{r}c_{I21}$, $m_{021}{}^{r}m_{I21}$,

amande blanche

$$c_{021}\ c_{021}{}^{r}c_{I21} \qquad\qquad \rightarrow c_{I21}$$

pétillant : $c_{011}{}^{r}c_{II11}$, $c_{I11}{}^{r}c_{II11}$, $m_{011}{}^{r}m_{II11}$, $m_{I11}{}^{r}m_{II11}$,

vin pétillant

$$m_{011}\ m_{011}{}^{r}m_{II11} \qquad\qquad \rightarrow m_{II11}$$

vin blanc pétillant

$$m_{011}\ m_{011}{}^{r}m_{I11}\ m_{I11}{}^{r}m_{II11} \rightarrow m_{II11}$$

bon : $c_{211}c_{011}{}^{\ell}$, $c_{211}c_{I11}{}^{\ell}$, $c_{211}c_{II11}{}^{\ell}$, …, $m_{211}m_{011}{}^{\ell}$, $m_{211}m_{I11}{}^{\ell}$, $m_{211}m_{II11}{}^{\ell}$, …

bon vin blanc pétillant

$$m_{211}m_{II11}{}^{\ell}\ m_{011}\ m_{011}{}^{r}m_{I11}\ m_{I11}{}^{r}m_{II11} \rightarrow m_{211}m_{II11}{}^{\ell}\ m_{II11} \rightarrow m_{211}.$$

A comment on our use of indices is warranted: Note that the types of *petite chat* or *chat petit* will not reduce to a simple type. Types which differ "only" by the value of an index, say c_{011} and c_{111}, are actually just as different as n and c_{012}. The use of indices is convenient, when defining the dictionary, i.e. when assigning types to the words of the language fragment. Each index in the subscript of a type symbol represents a "feature" of the concept, like gender, number, position of the adjective. The fact that a (sequence of) word(s) starts with a vowel or consonant should also reasonably be called a feature. This feature has here been placed in superscript, expressed by the presence or absence of the symbol ' (prime). The pregroup grammar does not include unification of features, but a good type assignment algorithm will.

For example, when checking whether *bon vin* is a well-formed construct of the language, we must try all possible type assignments of this sequence of two words until we hit one which reduces to the appropriate basic type. Only one of the possible types for *bon*, namely $m_{211}m_{011}{}^{\ell}$, will result in a string which reduces the type of *bon vin* to m_{211}. Note that another type for *bon* must be used in *bon vin blanc pétillant*. Type assignment can be made more efficient by keeping the variable h as long as possible instead of immediately replacing it by its possible values. The Meta-rule describes types for *bon* as $x_{211}x_{h11}{}^{\ell}$, for $h = 1,0,I,II,\ldots$, $x = c, m$. This corresponds to eight or more types, depending on the number of hierarchy classes. As a first step, the improved type assignment algorithm would assign

the string $x_{211}x_{h11}{}^{\ell}\,m_{011}$ to *bon vin* and at a second step make $x = m$ and $h = 0$. Hence, a simple calculation of equality of "features" is part of an efficient type assignment algorithm, by far exceeding the efficiency of the general recognition algorithm.

4.2. Determiners

The complete noun phrase, introduced by a possessive or demonstrative pronoun, definite or indefinite article, can be a subject or attribute or direct object. It has type n_{gn}. A complete noun phrase formed with a mass noun can be preceded by the proposition *de* and become a partitive complete noun phrase. Therefore we introduce new basic n_{gnq}, and require $n_{gnq} \to n_{gn}$, where $q \in \{1,2\}$. Here 1 stands for complete noun phrases formed with a mass noun, 2 stands for complete noun phrases formed with a count noun.

General complete noun phrase

Roughly speaking, names are complete noun phrases and so are nouns, with or without adjectives, if preceded by an article or a demonstrative or possessive pronoun.

Albert : n'_{111}

Marie : n_{211}

le : $n_{11}x_{11}{}^{\ell}$, $x = c,m$

les : $n_{g2}x_{g2}{}^{\ell}$, $x = c,c'$, $g = 1,2$

ce, mon, ton, son, notre, votre, leur : $n_{11q}x_{11}{}^{\ell}$, either $x = c, q = 2$ or $x = m, q = 1$

la, cette, ma, ta etc. : $n_{21q}x_{21}{}^{\ell}$, either $x = c, q = 2$ or $x = m, q = 1$

cette, mon, ton, etc. : $n_{21q}x'_{21}{}^{\ell}$, either $x = c, q = 2$ or $x = m, q = 1$

l' : $n_{g1q}x'_{g1}{}^{\ell}$, $g = 1,2$, either $x = c, q = 2$ or $x = m, q = 1$

cet : $n_{11q}x'_{11}{}^{\ell}$, either $x = c, q = 2$ or $x = m, q = 1$

ces, mes,…, : $n_{g21}x_{g2}{}^{\ell}$, $g = 1,2$, $x = c,c'$

un : $n'_{11q}x_{11}{}^{\ell}$, either $x = c,c', q = 2$ or $x = m,m'$ $q = 1$

une : $n'_{21q}x_{21}{}^{\ell}$, either $x = c,c', q = 2$ or $x = m,m'$ $q = 1$

The difference between the types of *le, les* and the other determiners lies in the fact that prepositions like *de, à* contract with *le, les* to yield a new word: *du,* (**de le*), *des* (**de les*), etc.

Recalling that $n'_{gnq} \to n'_{gn}$ and $m_{hgn} \to m_{gn}$, we use generalised contractions to analyse the following example:

un bon vin blanc

$$n'_{111}m_{11}{}^{\ell}\quad m_{211}\quad\quad \to n'_{111}m_{11}{}^{\ell}\, m_{11}\ \to n'_{111}\ \to\ n'_{11}$$

Note that these determiners yield complete noun phrases which can be a subject, object or attribute: *un bon vin blanc me plait, j'aime un bon vin blanc, c'est un bon vin blanc.*

Partitive complete noun phrase

   French has complete noun phrases formed with the partitive article, *du, de la, de l', de,* etc. Functioning as a partitive [9], *de* transforms an incomplete noun phrase into a complete one. This partitive noun phrase can be a direct object of a verb (*Il mange du pain*), attribute *(C'est du sable)* or even subject (*des enfants jouent dans la rue*), i.e. the partitive article is understood as an indefinite article. In everyday French, however, a noun phrase with the partitive article in the singular is rarely used as the subject of a sentence: **Du pain est sur la table*, *?De l'eau s'est infiltrée dans les fondements, *Du sable gêne l'engrenage* are replaced by *Il y a du pain sur la table. Il y a de l'eau qui s'est infiltrée. Il y a du sable qui gêne l'engrenage.*

   We introduce a new type $\hat{n}_{gn}$, $g=1,2;\, n=1,2$, together with a super-type $\hat{n}$, such that $\hat{n}_{gn} \to \hat{n}$. This is the type of complete noun phrases which generally will not be used as subject. The plural partitive article *des* transforms a plural count noun into a complete noun phrase. The same holds for the singular partitives *du, de la, de l'*, when preceding a mass noun phrase.

Hence the types

des : $\hat{n}_{g2}x_{g2}{}^{\ell},\ x=c,c'$ [10]

du   : $\hat{n}_{11}m_{11}{}^{\ell}$

de   : $\hat{n}_{g1}n_{g11}{}^{\ell}$

---

[9] i.e. which selects out of a mass or a group

[10] This implies that *des jolies fleurs* is considered as a complete noun phrase.

Examples

     (Je mange) des pommes   *(Je mange) du   pain*

       $\hat{n}_{22}c_{22}{}^{\ell}\ \ c_{022}\quad \rightarrow \hat{n}_{22}$ $\hat{n}_{11}m_{11}{}^{\ell}\ m_{011} \rightarrow \hat{n}_{11}$

     (Il vend)   du   vin   blanc

       $\hat{n}_{11}m_{11}{}^{\ell}\ m_{011}\ m_{011}{}^{r}m_{I11} \rightarrow \ \hat{n}_{11}$

The type of *du vin blanc* contains a critical triple and is identical to our running example $cb^{\ell}aa^{r}d$.

Note that the introduction of a new basic type for partitive complete noun phrases makes it possible to define different semantical interpretations for

  (1)   *Des gens vous demandent*.

  (2)   **Des nombres pairs sont divisibles par deux.*

  (3)   *Les nombres pairs sont divisibles par deux.*

The first sentence is generally accepted, see [Le bon usage], [Carlier], [Kleiber], whereas the second is rejected (because of the "wrong" meaning) and replaced by the third. Our analysis assigns different types to the noun phrases *des nombres pairs* and *les nombres pairs*, namely $\hat{n}_{12}$ and n_{12} . By an appropriate type of the French verb, it will therefore be possible to accept (1) and (3) and to reject (2).

5. Conclusion

We have shown that pregroup grammars are efficient and language independent from a computational view point, by giving a recognition algorithm running in time $o(n^3)$ with a constant which is independent of the number of symbols or rules of the grammar. One way how pregroup grammars gain in expressive power is by introducing a higher number of primitive categories (basic types) than do the more classical categorial grammars. To increase efficiency of type assignment, the basic types may be organized by features.   Ongoing work will show that unification of features can be used as a strategy combine type assignment and type checking in linear the time. The starting point for this is the linear "lazy" type checking algorithm obtained as a special case of the general algorithm.

References

[Barg Lamb]   Danièle Bargelli, Joachim Lambek, An algebraic approach to the French sentence structure, in P. de Groote, G. Morrill, C. Retoré (eds.), Logical Aspects of Computational Linguistics, pp. 95-109, LNAI 2099, Springer, 2001

[Buszkowski]   Wojciech Buszkowski, Lambek Grammars based on pregroups, in: P. de Groote, G. Morrill, C. Retoré (eds.), Logical Aspects of Computational Linguistics, LNAI 2099, Springer, 2001

[Carlier]   Anne Carlier, La Résistance des articles *du* et *des* à l'interprétation générique, in D. Amiot et al., editors, Le syntagme nominal syntaxe et sémantique, Artois Presses Université, 2001

[Cas-Lamb]   Claudia Casadio, An algebraic analysis of clitic pronouns in Italian, in P. de Groote et al. (eds), Logical aspects of computational linguistics, Springer LNAI 2099, Berlin 2001, 110-124

[CYK]   David Younger, Recognition and Parsing of Context-Free Languages in Time n3, Information and Control, 10:2, 1967

[Degeilh]   Linear tagging of agreement in the French verb phrase, preprint, Proceedings of Categorial Grammars 2004, Montpellier France, 7-11 June, 2004

[Earley]   Jay Earley, An efficient context-free parsing algorithm, Communications of the AMC, Volume 13, Number 2, pp 94-102, 1970

[Kleiber]   Georges Kleiber, Indéfinis: lecture existentielle et lecture partitive, in G. Kleiber et al., editors, Typologie des groupes nominaux, Presses Universités Rennes, 2001

[Kislak]   Alessandra Kislak, Pregroups versus English and Polish Grammar, in: New Perspectives in Logic and Formal Linguistics, Bulzoni Editore, Bologna, pp. 129-154, 2002

[Lambek 58]   Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1958

[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 03]   Joachim Lambek, A computational algebraic approach to English grammar, preprint, McGill University, Montreal, 2003, to be published in SYNTAX

[Lamb-Prel NP]   Joachim Lambek, Anne Preller, An algebraic approach to the German noun phrase, Linguistic Analysis, Vol. 31, 3-4, 2001

[Lamb-Prel SE]   Joachim Lambek, Anne Preller, An algebraic approach to the German sentence, Linguistic Analysis, Vol. 31, 3-4, 2001

[Le bon usage] Maurice Grevisse, André Goosse, Le bon usage, grammaire française, Duculot, 2001

[Math-Ling]   Fields Institute Summer School, Logic and Foundations of Computation, Workshop on Mathematical Linguistics, University of Ottawa, June 18-19, 2003

Danièle Bargelli, Joachim Lambek: "An algebraic approach to Arabic sentence structure"

Claudia Casadio, Joachim Lambek: "An algebraic approach to Latin sentence structure"

Kumi Cardinal: "An algebraic approach to Japanese sentence structure"

[Oehrle]   Richard Oehrle, A parsing algorithm for pregroup grammars, preprint, Proceedings of Categorial Grammars 2004, Montpellier France, 7-11 June, 2004

[Preller]   Anne Preller, Pregroups and linear processing of coordinate structures, preprint, Proceedings of Categorial Grammars 2004, Montpellier France, 7-11 June, 2004