Journal of Logic, Language and Information (2022) 31:327–388
https://doi.org/10.1007/s10849-022-09363-0

IDL-PMCFG, a Grammar Formalism for Describing Free Word Order Languages

François Hublet[1]

Accepted: 23 March 2022 / Published online: 29 April 2022

Abstract
We introduce *Interleave-Disjunction-Lock parallel multiple context-free grammars* (IDL-PMCFG), a novel grammar formalism designed to describe the syntax of free word order languages that allow for extensive interleaving of grammatical constituents. Though interleaved constituents, and especially the so-called hyperbaton, are common in several ancient (Classical Latin and Greek, Sanskrit...) and modern (Hungarian, Finnish...) languages, these syntactic structures are often difficult to express in existing formalisms. The IDL-PMCFG formalism combines Seki et al.'s parallel multiple context-free grammars (PMCFG) with Nederhof and Satta's IDL expressions. We define the semantics of IDL-PMCFGs and study their expressivity, proving that IDL-PMCFG extends both PMCFG and IDL-CFG (context-free grammars equipped with IDL expressions) and that IDL-PMCFG parsing is NP-hard. We then introduce COMPĀ, a programming language extending Ranta's Grammatical Framework (GF) and built as a high-level front-end formalism to IDL-PMCFG for practical grammar development. We present a parsing algorithm for IDL-PMCFG inspired by earlier Earley-style PMCFG parsing algorithms and Nederhof and Satta's IDL graphs and give a worst-case estimate of its complexity as a function of several metrics on IDL expressions, the size of the input and a new notion of the G-density of a language.

Keywords Grammar formalism · Free word order · Midly context-sensitive languages · PMCFG · Grammatical Framework

✉ François Hublet
francois.hublet@inf.ethz.ch

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland

1 Modelling and Parsing Free Word Order Languages: A Brief State of Affairs

1.1 The Challenge of Free Word Order

Since Kashket (1986)'s seminal contribution, developing models and parsing techniques for free word order languages has been an ongoing challenge for computational linguists. Whilst free word order phenomena are largely absent from modern Western languages such as English, they are frequent in ancient Indo-European languages such as Sanskrit (Schaufele 1991), Greek and Latin (Conrad 1965; Devine and Stephens 2006; Spevak 2010), in Finno-Ugric languages such as Hungarian (Kiss 1981) or Finnish (Kay and Karttunen 1984), but also in Australian (Kashket 1986; Austin 2001), Turkic (Hoffman 1995), and, to a certain extent, Slavic (Siewierska and Uhlirova 1998) and Germanic (Reape 1994) idioms. In morphologically rich languages, a certain level of word order freedom is generally present, which can range from simple relaxation of linear ordering contraints to genuine non-configurationality. Additionnally, loosening of common word order constraints is a frequent feature of literary, especially metrical, texts, in which prosodic, stylistic and expressive factors favor alternative and unusual word orderings.

At this point, it is worth mentioning that even the notion of *free word order* is, in itself, rather imprecise. Three different phenomena are generally qualified as such: (i) freedom in linear reordering or grammatical constituents as in *Today I walk—I walk today* (ii) discontinuous constituents that may span over a whole sentence; this common feature of e.g. German can also be demonstrated with English phrasal verbs in sentences such as *I* <u>*checked*</u> *this* <u>*out*</u> (iii) hyperbaton, i.e. interleaving of grammatical constituents as they frequently occur for instance in Classical Latin: *cetera labuntur celeri caelestia motu*[1] ('the other heavenly [bodies] move quickly', litt. 'the-other move quick heavenly movement'). This last and, by most aspects, most complex phenomenon produces crossing dependencies between constituents. Classical Latin, which provides innumerable examples of this, will serve as a reference for further investigation, but similar patterns can also be exhibited in Ancient Greek, Sanskrit, Old Norse, Slavic and Finno-Ugric languages, among others.

Context-free grammars (CFGs), introduced by Noam Chomsky in the 1950s, can be considered the *de facto* baseline of most generative grammar formalisms in both computer science and linguistics. Nevertheless, CFGs, unlike many dependency grammar formalisms, turned out to be unable to describe certain syntactic phenomena occurring in the grammar of natural languages, especially those involving free constituent order or discontinuous constituents. These limitations fostered the development of new, non context-free formalisms better suited to describe natural language: indexed grammars (Aho 1968), immediate dominance/linear precedence grammars (ID/LP) (Pullum 1982; Shieber 1984), tree-adjoining grammars (TAG) (Vijayashanker and Joshi 1988), parallel multiple context-free grammars (PMCFG) (Seki et al. 1991), affix grammars over a fixed lattice (AGFL) (Koster 1991), positive range concatenation grammars (PRCG) (Boullier 1998), among others. Following Chomsky (1956),

[1] Cicero, *Aratus*, 358, cited by Conrad (1965).

these formalisms can all be classified as Type-1 grammars, and the languages they generate are generally referred to as context-sensitive. Most of the effort focussed on the development of so-called *midly context-sensitive* formalisms. A complete survey of the most common non context-free formalisms and their use in computational linguistics can be found in Kallmeyer (2010).

With rather strict word order languages accounting for a significant part of the available digital corpora and potential application fields, computational linguists, many of whom are native speakers of one of these idioms, may have been tempted to address the grammatical modelling of free word order languages with tools chiefly designed to describe English or similar languages. These tools rarely integrate an operator allowing for arbitrary constituent order, let alone for interleaving, since such operators come with a high computational cost that can and must be avoided when parsing free word order languages. This is especially true as regards multilingual parsing, translation or text generation systems that would have added support for some of the above languages at a later stage of their development. From all grammatical formalisms described above, only ID/LP can easily encode hyperbaton, but does not provide support for discontinuous constituents.

Note that this paper does *not* make a theoretical claim that none of the existing mildly context-sensitive formalisms is expressive enough, from a theoretical viewpoint, to encode free word order phenomena observed in natural languages. There are in fact good reasons to think that some of them are. In practice, if we assume that interleaving phenomena always have a finite depth, we can encode hyperbatic phenomena through a finite, yet exponential, number of context-free rules; recent theoretical results (Ho 2018) have shown that even without a finite-depth assumption, hyperbaton without copy is still mildly context-sensitive. What this paper *does* observe, however, is that we lack a general grammar description framework with built-in support for free word order phenomena, in which describing e.g. Classical Latin syntax requires neither an exponential inflation in the number of rules compared to the fixed word order case nor a complex conversion process. We lack a framework that would allow us to describe free word order syntax *as linguists or grammarians would do*, e.g. by defining single attachment rules that do not necessarily impose ordering constraints.

Early attempts to design grammatical formalisms for free word order languages have not led to the development of general-purpose tools; nor were they designed to provide cross-lingual interoperability with fixed word order languages. Covington (1990)'s approach, whose applications to parsing a "tiny subset of Latin" were explored by Koch (1993), relies on dependency rather than phrase structure grammar, which both authors consider less suited to addressing free word order phenomena. Dependency- and constraints-based methods have also been implemented by Bharati and Sangal (1993) for Indian languages, building on notions from Pāṇinian grammar. Though underlying dependency relations between words are indeed the real issue while describing the syntax of free word order languages, we do not believe that this point of view should be deemed irreconciliable with the traditional structured approaches to grammar writing, that involve clear-cut constituents.

We are indeed looking for a formalism that would allow us to conveniently describe the syntax of free word order languages, and that could be used to produce wide-coverage, modular grammars in the style of the Ressource Grammar Library

(Ranta et al. 2009). In addition to providing native support for free word order language, the new framework would still be able to encode standard fixed order rules; ideally, it would be built as a "free word order extension" of on existing framework, in order to capitalize on past efforts and guarantee compatibility with existing fixed word order grammars. A new formalism fulfilling these requirements, which we will introduce and study in Sect. 2, is called *Interleave-Disjunction-Lock parallel multiple context-free grammars* or IDL-PMCFG.

Another essential factor to take into account when designing a grammatical formalism is its suitability for practical implementation of wide-coverage grammars. One way to ensure that users can easily define and use their own grammar models is to provide a complete front-end syntax for grammatical description in the form of a special-purpose programming language. In these regards, we built on Ranta (2011)'s Grammatical Framework (GF) and Nederhof and Satta (2004)'s IDL expressions to elaborate our own grammar description system, *COMPĀ*, whose syntax extends so-called *context-free GF* (Ljunglöf 2004) with some new operators to encode interleaving, disjunction and locking of constituents. High-level COMPĀ code is compiled into a low-level IDL-PMCF grammar that can be used directly for parsing. COMPĀ and its compiler are introduced in Sect. 3; the parsing algorithm itself is presented and studied in Sect. 4.

Before we proceed with the description of our formalism, a short look at precise linguistic facts behind extensive free word order can help us identify the exact features we are looking for.

1.2 Towards a Natural Account of Free Word Order Syntax: The Case of Classical Latin

A language with considerable freedom of word order, Classical Latin presents many syntactic phenomena alien to most modern Western European languages. By looking at the few typical aspects of Latin syntax, we shall see in this section which kind of features our desired framework should have in order to be able to concisely encode the syntactic phenomena at play in free word order languages in general, and in Classical Latin in particular.

1.2.1 Hyperbaton and Interleaved Constituents

As Devine and Stephens (2006) puts it, "[p]hrasal discontinuity, traditionally called hyperbaton in Classical studies, is perhaps the most distinctively alien feature of Latin word order". Hyperbaton is a very general, transcategorial phenomenon that can occur whenever a syntactic constituent is non-contiguous. Danckaert (2017) emphasizes that modern research has shifted away from the opposition of regular vs. exceptional word orders as it is found for example in Marouzeau (1922); still, recent transformational approaches have relied on some kind of default word order to distinguish non-emphatic from non-emphatic word orders. This might be totally justified when pragmatic information is available, provided that, in the words of Devine and Stephens (2006), "[t]he syntax is massaged to provide for a simple and direct translation into a pragmatically structured meaning". Unfortunately, such information is generally not available in usual parsing contexts.

In particular, the ante- or postposition of adjectives and genitive modifiers in Classical Latin does not obey general syntactic rules (Devine and Stephens 2006). Statistical patterns may vary from word to word, with patterns rarely uniformly spreading over whole semantic lexical categories. Not surprisingly, discontinuous adjective and genitive attachement represents an overwhelming majority of all instances of hyperbata. In verse, where discontinuity is the standard rather than the exception, Conrad (1965) has shown it to be a characteristic feature of a long Greco-Roman poetic tradition dating back to the oral tradition of the Homeric times, influenced by the Roman taste for phenomena such as the clash of ictus and accent on the fourth foot of the hexameter. Latin poets made such an extensive use of the device that in Horace, we find stanzas with three crossing attribute dependencies.[2]

In this context, there is no reason to deny hyperbaton its status as a standard, independent feature of Classical Latin; as parsing systems do not have access to pragmatic information, and since hyperbaton is extremely common even in simple prose, **we need to be able to formulate general** adjective attachment **rules that**, within the clause, **relaxes all constraints on both linear order and intervention of other constituents**.

1.2.2 Locking of Clauses and Prepositional Phrases

One seemingly absolute constraint on word reordering in Classical Latin concerns the impossibility of so-called 'long hyperbata' between finite clauses. 'Long hyperbata' are defined in Devine and Stephens (2006) as hyperbata that involve the extraction of a word from one clause to another; 'short hyperbata', on the other hand, are hyperbata that allow for interleaving words only within the bounds of a given clause. **We must be able to express that finite clauses *generally* need to be 'locked'**, i.e. protected against interleaving with other clauses.

We only say 'generally', since verse texts provide well-known counter-examples to this rule,[3] showing that mixing of material from different finite clauses was not altogether impossible in poetic contexts. Moreover, it must be noted that this general exclusion of long hyperbata in finite clauses does not generalize to non-finite (infinitive and participial) clauses, which can be freely interleaved.

Another important issue, especially in verse, is that of the position of the subordinator not at the beginning, but within the clause, that has been extensively studied by Marouzeau (1949). Bortolussi (2006) has emphasized the high occurence frequency and expressive value of this so-called *traiectio*, which leads to the subordinator appearing (at least) second in the clause. Yet, an almost absolute rule that opposes rightward movement of subordinators is that a subordinator cannot stand last within the clause it introduces. Therefore, **we** still **need to be able to restrict (linear) freedom of word order in certain cases**.

Finally, another instance of locking with an additional constraint on word order occurs in the context of prepositional phrases: while all but one element of the prepo-

[2] See Horace, *Carm.* 1.9.21-22, cited by Marouzeau (1922).

[3] See Horace, *Sat.*, 1.5.72, cited by Marouzeau (1922).

sitional phrase might be arbitrary interleaved within the clause, at least one element (not necessarily the head) must be placed directly after the preposition. To account for this type of syntactic limitation, a combination of mostly free word order with targeted locking and linear constraints is again required.

1.2.3 Multiple Fields and Features

General-purpose grammar description systems such as Grammatical Framework (Ranta 2004) make an extensive use of records and fields in order to store the various forms of a word, parts of discontinuous grammatical constituents or handle reduplication phenomena. As our goal is to be able to describe Classical Latin syntax as generally as possible and we may want to keep some interoperability with existing frameworks, **records and fields are required in practice**.

There is, however, no obvious reason why we should require copying to be available in order to describe Classical Latin. Allowing copy in our framework can be desirable in order to account for specific syntactic phenomena in other natural languages (see below), because copying is a general phenomenon in language (Kobele 2006), or to preserve compatability with existing tools such as Grammatical Framework. But this is a design decision independent from the specific characteristics of Latin syntax, whose goal is not to stick closely to the formal requirements of Latin syntax, but rather to preverse some general linguistic expressiveness. As it makes sense to think of a new framework as having to match the needs of free word order languages in general and not of Classical Latin exclusively, **we will want to allow copy operations in our formalism**.

1.2.4 Summary

The above discussion suggests five characteristics that a grammatical formalism designed to enable a straightforward description of the syntax of free word order languages such as Classical Latin should have: operators to interleave grammatical constituents, lock phrases and restrict reorderings of constituents; a record and fields system; and, finally, and maybe less importantly, a support for copy operations.

Notations We will use the following conventions:

- $\mathbb{N}^+$ denotes $\mathbb{N}\backslash\{0\} = \{1, 2, \ldots\}$;
- Symbol ε denotes the empty word, while $\underline{\varepsilon}$ and $\diamond$ (‘diamond’) are special symbols;
- All alphabets Σ used in this paper are assumed not to contain the symbols $\underline{\varepsilon}$ and $\diamond$;
- For $(a, b) \in \mathbb{N}^2$, $[\![a, b]\!]$ denotes the set $\{a, a+1, \ldots, b-1, b\}$ and $[\![a, b[\![$ the set $\{a, a+1, \ldots, b-1\}$;
- For any set S, $\mathscr{P}(S)$ denotes the power set (set of subsets) of S and $\mathscr{P}_f(S)$ the set of finite subsets of S;
- For any set S, $|S|$ denotes the cardinal of S;
- For any sets S, T, $S \rightharpoonup T$ denotes a partial function from S to T;
- For all $f : S \rightharpoonup T$, $\mathscr{D}(f)$ denotes the domain of f;

- Let Σ be an alphabet and $t \in \Sigma^*$ a word on this alphabet. Then $|t|$ denotes the length of t. Furthermore, for all $p \in \mathscr{P}(\llbracket 1, |t| \rrbracket)$, t_p denotes the subword of t formed by extracting the symbols at positions p in t. For example, on $\Sigma = \{a, \ldots, z\}$:

$$\begin{aligned} alphabet_{\{1,8\}} &= at \\ alphabet_{\{2,5,6\}} &= lab \\ alphabet_{\{6,7,8\}} &= bet; \end{aligned}$$

- Rules in grammars are written using the following functional notation

$$A_1 \to \cdots \to A_n \to B : a_1, \ldots, a_n \mapsto e$$

which reads "given an item a_1 of category A_1, …, an item a_n of category A_n, an item of category B can be produced which is equal to expression e"; expression e depends on the current formalism but will usually contain instances of a_1, …, a_n;

2 Introducing IDL Parallel Multiple Context-Free Grammars (IDL-PMCFG)

2.1 IDL Expressions

IDL (*Interleave-Disjunction-Lock*) expressions, introduced by Nederhof and Satta (2004), are a family of regular expressions tailored to describe and parse natural language sentences. Since they do not allow for the use of nonterminal symbols, Nederhof and Satta's original IDL expressions are no *grammars* and can therefore only be used to describe specific (finite) families of utterances; a single IDL expression cannot encode a complex language model. However, they already include everything needed to account for free constituent order, hyperbata and their respective limitations. The definitions below closely follow those of the original paper.

Definition 1 (*IDL expression*) Let Σ be a finite alphabet. An IDL expression e over Σ is defined inductively as follows:

$$\begin{aligned} e := a \quad & \forall a \in \Sigma \cup \left\{\underline{\varepsilon}\right\} \\ \mid e' \cdot e'' & \\ \mid \times\left(e'\right) & \\ \mid \vee\left(e_1, \ldots, e_n\right) \quad & \forall n \in \mathbb{N}^+ \\ \mid \,||\left(e_1, \ldots, e_n\right) \quad & \forall n \in \mathbb{N}^+. \end{aligned}$$

Note that, unlike usual regular expressions dealing with character strings, IDL expressions used in typical computational linguistics applications use an alphabet Σ composed of full words (tokens), that are to be combined into grammatical constituents and sentences. Therefore, throughout this document, the word *string* should be understood as a shortcut for 'token list', an the word *set of strings* as a shortcut for 'set of token lists'. The informal semantics of the constructors, that all act on sets of strings, is as follows:

– The dot represents standard concatenation;
– Disjunction has its usual semantics as set union;
– The interleave operator || allows for arbitrarily mixing tokens contained in n strings, as long as the relative ordering within each initial string is preserved in the final string.
– The lock operator $\times$ prevents a string from being divided into several substrings by an instance of the interleave operator.

This can be illustrated by the following variations on the Latin sentence *ciuis Romanus sum* ('I am a Roman citizen', litt. 'citizen Roman am'):

$$\begin{aligned}
\texttt{ciuis}\cdot\texttt{Romanus}\cdot\texttt{sum} &\longrightarrow \{\ \texttt{"ciuis Romanus sum"}\ \}\\
\vee(\texttt{ciuis},\texttt{Romanus},\texttt{sum}) &\longrightarrow \{\ \texttt{"ciuis"},\texttt{"Romanus"},\texttt{"sum"}\ \}\\
||(\texttt{ciuis},\texttt{Romanus},\texttt{sum}) &\longrightarrow \{\ \texttt{"ciuis Romanus sum"},\texttt{"ciuis sum Romanus"},\\
&\qquad\ \texttt{"Romanus ciuis sum"},\texttt{"Romanus sum ciuis"}\\
&\qquad\ \texttt{"sum ciuis Romanus"},\texttt{"sum Romanus ciuis"}\ \}\\
||(\texttt{ciuis}\cdot\texttt{Romanus},\texttt{sum}) &\longrightarrow \{\ \texttt{"ciuis Romanus sum"},\texttt{"ciuis sum Romanus"},\\
&\qquad\ \texttt{"sum ciuis Romanus"}\ \}\\
||(\times(\texttt{ciuis}\cdot\texttt{Romanus}),\texttt{sum}) &\longrightarrow \{\ \texttt{"ciuis Romanus sum"},\texttt{"sum ciuis Romanus"}\ \}
\end{aligned}$$

To formally define the language of an IDL expression, we first need to introduce the primitives lock and comb as done in Nederhof and Satta (2004):

Definition 2 (*Primitives* lock *and* comb)

1. Let lock be the only monoid homomorphism over $((\Sigma\cup\{\diamond\})^*,\cdot)$ such that $\mathsf{lock}_{|\Sigma}=\mathrm{id}_{|\Sigma}$ and $\mathsf{lock}(\diamond)=\varepsilon$.
2. Let comb and comb′ be functions from $\left((\Sigma\cup\{\diamond\})^*\right)^2$ to $\mathscr{P}\left((\Sigma\cup\{\diamond\})^*\right)$ defined inductively by:

$$\mathsf{comb}(x,y)=\mathsf{comb}'(x,y)\cup\mathsf{comb}'(y,x)$$

$$\mathsf{comb}'(x,y)=\begin{cases}\{x\diamond y\} & \text{if there is no } \diamond \text{ in } x\\ \{x'\diamond y'\mid y'\in\mathsf{comb}(x'',y)\} & \text{if } x \text{ is of the form } x'\diamond x'' \text{ with no } \diamond \text{ in } x'\end{cases}.$$

Informally, the symbol $\diamond$ represents positions at which words can be interleaved into the current substring. Such $\diamond$ symbols are inserted into the current string by each concatenation or interleave operation: by default, every word boundary is a place where a new word can be inserted. The lock primitive erases such symbols in each string of the input set, thus preventing any interleaving within the enclosed substrings. The comb primitive produces the set of all strings that can be obtained by interleaving contiguous substrings of the two input string at positions marked by a $\diamond$.

Since comb produces all possible interleavings of two input strings, it is clearly associative and commutative. We can see comb as an $n$ary operator for all n, and write $\mathsf{comb}_{i=1}^{n}a_i := \mathsf{comb}(a_1,\mathsf{comb}(a_2,\ldots\mathsf{comb}(a_{n-1},a_n)\ldots))$.

We can now define the language of an IDL expression, which exactly matches the mechanics exposed and demonstrated above:

Definition 3 (*Language of an IDL expression*) Let Σ be an alphabet. For any IDL expression e over Σ, language $\mathrm{L}(e)$ is given by

$$\mathrm{L}(e) = \sigma(\times(e))$$

where for any IDL expression e over Σ, $\sigma(e)$ is a subset of $\mathscr{P}\left((\Sigma \cup \{\diamond\})^*\right)$ that is defined inductively as follows:

$$\begin{aligned}
\sigma(\underline{\varepsilon}) &= \{\varepsilon\} \\
\sigma(a) &= \{a\} && \forall a \in \Sigma \\
\sigma(e' \cdot e'') &= \left\{w' \diamond w'' \mid (w', w'') \in \sigma(e') \times \sigma(e'')\right\} \\
\sigma(\times(e')) &= \mathsf{lock}(\sigma(e')) \\
\sigma(\vee(e_1, \ldots, e_n)) &= \bigcup_{i=1}^{n} \sigma(e_i) \\
\sigma(\|(e_1, \ldots, e_n)) &= \mathsf{comb}_{i=1}^{n} \sigma(e_i)\,.
\end{aligned}$$

To see why IDL expressions are well-suited to describe grammatically valid reorderings of utterances in free word order languages, consider the following example from Latin: *Marcus cum amico caro ambulat* (‘Marcus walks with his dear friend’, litt. ‘Marcus with friend dear walks’). For a permutation of the five above words to be considered valid in Classical Latin verse, the only condition to be met is that *cum* (‘with’) must stand immediately before either *amico* (‘friend’, ablative singular) or *caro* (‘dear’, ablative singular masculine). Besides this single constraint, the order of constituents is free, and the verb modifier *cum amico suo* might even be disjoint. This means that even heavily reordered utterances such as *amico Marcus ambulat cum caro* should be considered grammatical, as similar structures are, indeed, well documented. Now, this seemingly unusual syntactic constraint is surprisingly easy to express in terms of an IDL expression:

$$\|(\mathtt{Marcus}, \vee(\|(\times(\mathtt{cum}\cdot\mathtt{amico}), \mathtt{caro}), \|(\times(\mathtt{cum}\cdot\mathtt{caro}), \mathtt{amico})), \mathtt{ambulat})\,.$$

Of course, IDL expressions alone cannot provide much more than *ad-hoc* solutions for a set of specific utterances. In order for their expressive power to be used for general language description, they must be integrated into a complete grammatical formalism.

2.2 Parallel Multiple Context-Free Grammars

It was already to deal with discontinuous constituents in natural language that Seki et al. (1991) defined parallel multiple context-free grammars (PMCFG), whose definition is given below. Parallel multiple context-free grammars extend context-free grammars by manipulating tuples of strings instead of strings. Each category of a PMCFG is assigned a dimension (a tuple size). Every production consumes a number of named argument tuples of fixed categories and produces a new tuple. Each element of this tuple is the concatenation of an arbitrary number of terminals and (nonterminal, index) pairs, which uniquely identify a field of one the arguments. The start category, usually denoted by S, defines the grammar's language, and has therefore dimension 1.

We have the following typical example:

Lemma 1 *Language* $\mathrm{L}_{3n} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ *on* $\Sigma_3 = \{a, b, c\}$ *is in* PMCFL.

Proof The following PMCFG grammar matches L_{3n}

$$\begin{aligned} T &\to S : t \mapsto \langle t\,[0] \cdot t\,[1] \cdot t\,[2] \rangle \\ T &\to T : t \mapsto \langle a \cdot t\,[0]\,, b \cdot t\,[1]\,, c \cdot t\,[2] \rangle \\ &\quad T : \langle \underline{\varepsilon}, \underline{\varepsilon}, \underline{\varepsilon} \rangle \end{aligned}$$

where, following usual programming conventions, we start indexing at 0 and write (nonterminal, index) pairs as nonterminal[index]. □

How does this grammar define the above language? First, it states that a (one-dimensional) tuple of type S can be produced from a (three-dimensional) tuple t of type T by concatenating the three fields of t; then, that a tuple of type T can be produced in either of two ways: either it is generated from another tuple t of type T by appending a, b and c respectively at the beginning of each of the fields, or is equal to $\langle \varepsilon, \varepsilon, \varepsilon \rangle$. It is straightforward to see that T matches L_{3n}, thus yielding the expected behavior for S.

The formal definition of PMCFG, slightly adapted from Seki et al. (1991), is as follows:

Definition 4 (*PMCF grammar*) A PMCF grammar (or PMCFG) is a sextuple

$$G = (N, \delta, \Sigma, F, P, S)$$

where

1. N is a finite set of nonterminal symbols (also called categories);
2. $\delta : N \to \mathbb{N}$ maps each nonterminal symbol A to its dimension $\delta\,(A)$;
3. Σ is a finite set of terminal symbols disjoint with N;
4. F is a finite set of functions such that for all $f \in F$, there exists $a\,(f) \in \mathbb{N}$, called arity of f, as well as $a(f)$ integers $d_1(f), \ldots, d_{a(f)}(f)$ encoding the dimensions of the $a(f)$ arguments of f, and an integer $r(f)$ encoding the dimension of the image of f, such that the signature of f is
$$\left(\Sigma^*\right)^{d_1(f)} \times \cdots \times \left(\Sigma^*\right)^{d_{a(f)}(f)} \to \left(\Sigma^*\right)^{r(f)};$$
5. For any $f \in F$, letting $\rho := r\,(f)$, f is of the form
$$s_1, \ldots, s_{a(f)} \mapsto \left\langle \alpha_{11} s_{\beta_{11}\gamma_{11}} \alpha_{12} s_{\beta_{12}\gamma_{12}} \ldots \alpha_{1\delta_1}, \ldots, \alpha_{\rho 1} s_{\beta_{\rho 1}\gamma_{\rho 1}} \alpha_{\rho 2} s_{\beta_{\rho 2}\gamma_{\rho 2}} \ldots \alpha_{\rho \gamma_\rho} \right\rangle$$

where all δ_i, β_{ij} and γ_{ij} are integers with $\beta_{ij} \leq a(f)$ and $\gamma_{ij} \leq d_{\beta_{ij}}(f)$, and $\alpha_{ij} \in \Sigma^*$ for all i, j. In other terms, every component of the tuple produced by f is obtained via arbitrary concatenation of symbols from Σ and components of f's arguments;
6. For $q \in \mathbb{N}$, let F_q denote the subset of all functions of arity q in F;
7. P, called the set of productions or rules, is a finite subset of $\bigcup_{q\in\mathbb{N}} \left(F_q \times N^{q+1}\right)$ such that for all $q \in \mathbb{N}$ and $\left(f, A_1, \ldots, A_{q+1}\right) \in P$, we have $d_k(f) = \delta(A_k)$ for every $k \in \{1, \ldots, q\}$ as well as $r(f) = \delta\left(A_{q+1}\right)$, i.e. the dimensions of the arguments (resp. of the image) of f match the dimensions of the categories on the left (resp. right) side of the production;
8. $S \in N$ is the start symbol, of dimension $\delta(S) = 1$.

Note that according to the previous definition, a PMCFG $G = (N, \delta, \Sigma, F, P, S)$ such that $\delta(N) = \{1\}$ (i.e. a PMCFG whose categories have dimension as most 1) is exactly a CFG.

Finally, we define the language of a parallel multiple context-free grammar as follows:

Definition 5 (*Language of a PMCFG*) Let $G = (N, \delta, \Sigma, F, P, S)$ be a PMCF grammar. Let $m = \max_{A\in N} \delta(A)$. We define a big-step derivation relation $\hat{\rightarrow}$ on $N \times \bigcup_{i=0}^{m} (\Sigma^*)^i$ inductively as follows:

For all $\left(A, \left\langle t_1, \ldots, t_{\delta(A)}\right\rangle\right) \in N \times (\Sigma^*)^{\delta(A)}$, we have $A \hat{\rightarrow} \left\langle t_1, \ldots, t_{\delta(A)}\right\rangle$ if, and only if, there exists a production $\left(f, A_1, \ldots, A_{a(f)}, A\right) \in P$ and strings $\left(s_{ij}\right)_{i\leq a(f), j\leq d_i(f)}$ such that the two following conditions are met:

1. For all integers $i \leq a(f)$, $j \leq d_i(f)$, $A_{ij} \hat{\rightarrow} s_{ij}$;
2. $f\left(s_1, \ldots, s_{a(f)}\right) = \left\langle t_1, \ldots, t_{\delta(A)}\right\rangle$.

The language recognized by G is defined as

$$\mathrm{L}(G) = \left\{s \in \Sigma^* \mid S \hat{\rightarrow} s\right\}$$

and call PMCFL (resp. CFL) the set of all languages that are recognized by at least one PMCFG (resp. CFG).

Let us give an example of how PMCFG extends the expressivity of CFG. We will use the following well-known example:

Lemma 2 *Language* L_{3n} *defined in Lemma 5 is not context-free.*

Proof This is a classical result whose proof (usually using the pumping lemma) can be found e.g. in Hopcroft et al. (2013). □

This lemma, combined with Lemma 5, results in the following strict inclusion:

Proposition 1 CFL $\subsetneq$ PMCFL.

Through its use of tuples, PMCFG provides a handy way to handle discontinuous constituents. Various parts of the linearization of a constituent can be stored in different fields, and later on integrated into a larger phrase. Since the same argument can appear an arbitrary number of times in the right hand side of any production, general PMCFGs can also define reduplication phenomena as encountered e.g. in short Swiss German verbs (Lötscher 1993), Indonesian plurals (Dalrymple and Mofu 2011) or Telugu distributives (Balusu 2006). The formalism has proved efficient as a parsing front-end for context-free GF (Ljunglöf 2004; Angelov et al. 2009; Ljunglöf 2012). Nevertheless, expressing free order of constituents or interleaving of constituents is not easy in PMCFG. Until 2018, it was not even known whether this was possible. Although the answer is now known to be positive (Ho 2018), there is still no convenient way to concisely express the interleaving of groups, since PMCFG lacks a specific operator for this type of reordering. One way to overcome this difficulty is to define all legal orderings manually and pass them as arguments to the corresponding rule; this technique has been demonstrated by Lange (2017) in the case of Latin.

2.3 Bringing Together IDL Expressions and PMCFG: IDL-PMCFG

The conclusions of the last two subsections suggest that we combine IDL expressions and parallel multiple context-free grammars into a single formalism that can handle discontinuous constituents and copy (as did PMCFG) as well as free constituent order and hyperbaton (as did IDL expressions). This formalism is IDL-PMCF grammars (IDL-PMCFG), whose definition is given below. Although the complexity of the membership problem of both IDL expressions and PMCF grammars is polynomial, this is not the case of their combination: we will see in this subsection that parsing IDL-PMCF grammars is NP-hard, which as an important corrolary (Theorem 1) implies that IDL-PMCF provides a strict extension of PMCFG unless P = NP.

In IDL-PMCF grammars, productions are defined not as concatenations, but as IDL expressions of terminals and (nonterminal, index) pairs. Instead of tuples of strings, *tuples of sets of strings* are now the basic data type manipulated by the various rules. The $\diamond$ symbol, which marks positions at which new words can be interleaved into the current string, is added to the alphabet.

Definition 6 (*IDL-PMCF grammar*) An IDL-MCF grammar (or IDL-PMCFG) is a sextuple

$$G = (N, \delta, \Sigma, F, P, S)$$

where

1. N is a finite set of nonterminal symbols (also called categories);
2. $\delta : N \to \mathbb{N}$ maps each nonterminal symbol A to its dimension $\delta(A)$;
3. Σ is a finite set of terminal symbols disjoint with N;
4. F is a finite set of functions such that for all $f \in F$, there exists $a(f) \in \mathbb{N}$, called arity of f, as well as $a(f)$ integers $d_1(f), \dots, d_{a(f)}(f)$ encoding the dimensions of the $a(f)$ arguments of f, and an integer $r(f)$ encoding the dimension of the image of f, such that the signature of f is

$$\left(\mathscr{P}\left(\overline{\Sigma}\right)\right)^{d_1(f)} \times \cdots \times \left(\mathscr{P}\left(\overline{\Sigma}\right)\right)^{d_{a(f)}(f)} \to \left(\mathscr{P}\left(\overline{\Sigma}\right)\right)^{r(f)}$$

where $\overline{\Sigma} = (\Sigma \cup \{\diamond\})^*$;

5. For any $f \in F$, there exist IDL expressions $e_1, \ldots, e_{r(f)}$ over $\Sigma \cup \left\{X_{ij} \mid i \leq a(f), j \leq d_i(f)\right\}$, where the X_{ij} are fresh variable symbols, such that f is of the form

$$s_1, \ldots, s_{a(f)} \mapsto \bigcup_{(w_1,\ldots,w_{a(f)}) \in s_1 \times \cdots \times s_{a(f)}} \left\{\left\langle x_1\left[X_{ij} := w_{ij}\ \forall i, j\right], \ldots, x_{r(f)}\left[X_{ij} := w_{ij}\ \forall i, j\right]\right\rangle \mid (x_1, \ldots, x_{r(f)}) \in \sigma(e_1) \times \cdots \times \sigma\left(e_{r(f)}\right)\right\}.$$

Function f now produces a set of tuples of length $r(f)$, which are derived in three steps: (1) for each $i \leq a(f)$, choose one tuple w_i in each set s_i (2) for $k \leq r(f)$, choose an x_k in each $\sigma(e_k)$ (3) for all i, j, substitute w_{ij} for X_{ij} in each x_k.

6. For $q \in \mathbb{N}$, let F_q denote the subset of all functions of arity q in F;
7. P, called the set of productions or rules, if a finite subset of $\bigcup_{q\in\mathbb{N}}\left(F_q \times N^{q+1}\right)$ such that for all $q \in \mathbb{N}$, for all function $f \in F$ and categories $A_1, \ldots, A_{q+1} \in N$ such that $\left(f, A_1, \ldots, A_{q+1}\right) \in P$, we have $d_k(f) = \delta(A_k)$ for every $k \in \{1, \ldots, q\}$ as well as $r(f) = \delta\left(A_{q+1}\right)$, i.e. the dimensions of the arguments (resp. of the image) of f match those of the categories on the left (resp. right) side of the production;
8. $S \in N$ is the start symbol, of dimension $\delta(S) = 1$.

Just as a parallel multiple context free grammar with $\delta(N) = \{1\}$ is a context-free grammar, we define IDL-CFGs as follows:

Definition 7 (*IDL-CF grammar*) An IDL-MCF grammar $G = (N, \delta, \Sigma, F, P, S)$ such that $\delta(N) = \{1\}$ is called an IDL-CF grammar (or IDL-CFG).

IDL context-free grammars are of essential theoretical interest. As we will come to evaluate the expressivity gain obtained by replacing simple concatenation by IDL expressions, it will be important to single out the contributions of both the PMCFG formalism and IDL expressions to the extension of the class of languages can that can be described by IDL-PMCFGs. Hence, comparing the expressivity of PMCFG to that of IDL-(PM)CFG, as we will do it in Sect. 2.4, shall inform us more thoroughly about the complementarity of the two approaches we combined.

Finally, the language matched by a given IDL-PMCFG can now be defined:

Definition 8 (*Language of an IDL-PMCFG*) Let $G = (N, \delta, \Sigma, F, P, S)$ be an IDL-PMCF grammar. Let $m = \max_{A\in N} \delta(A)$. We define a big-step derivation relation $\to$ on $N \times \bigcup_{i=0}^{m}\left(\mathscr{P}\left(\overline{\Sigma}\right)\right)^i$ inductively as follows:

For all $\left(A, \left\langle t_1, \ldots, t_{\delta(A)}\right\rangle\right) \in N \times \left(\mathscr{P}\left(\overline{\Sigma}\right)\right)^{\delta(A)}$, we have $A \to \left\langle t_1, \ldots, t_{\delta(A)}\right\rangle$ if, and only if, there exists a production $\left(f, A_1, \ldots, A_{a(f)}, A\right) \in P$ and sets of strings $\left(s_{ij}\right)_{i\leq a(f), j\leq d_i(f)}$ such that the two following conditions are met:

1. For all integers $i \leq a(f)$, $j \leq d_i(f)$, $A_{ij} \to s_{ij}$;
2. $f\left(s_1, \ldots, s_{a(f)}\right) = \left\langle t_1, \ldots, t_{\delta(A)}\right\rangle$.

Finally, we use the lock function introduced in Definition 3 to define the language recognized by G as

$$\mathrm{L}(G) = \bigcup_{\substack{t \in \mathscr{P}(\overline{\Sigma}) \\ S \to t}} \{\mathsf{lock}(s) \mid s \in t\}$$

and call IDLPMCFL (resp. IDLCFL) the set of all languages that are recognized by at least one IDL-PMCFG (resp. IDL-CFG).

Note that the $\diamond$ symbols are only erased from the output in the last step of the definition of $\mathrm{L}(G)$, after the set all strings that can be derived from the start symbol S has been retrieved. If they had been erased each time a production was used, interleaving constituents that are not arguments of the same rule would have been impossible, and constituents obtained from any production would have been locked.

Let us give an example of this. Consider the following IDL-CFG G_{abcd} on $\Sigma = \{a, b, c, d\}$ (when describing IDL-CFG grammars, we shall omit the [0] indexes identifying the first field of every argument):

$$\begin{gathered} Q \to R \to S : q, r \mapsto \|(q, r) \\ Q : a \cdot b \\ R : c \cdot d. \end{gathered}$$

Clearly, $\mathrm{L}(G_{abcd}) = \{abcd, acbd, acdb, cabd, cadb, cdab\}$. Consider the derivation tree for $acbd$:

$$\cfrac{\cfrac{Q \to \{a \diamond b\} \qquad R \to \{c \diamond d\}}{\exists s \ni a \diamond c \diamond b \diamond d \text{ s.t. } S \to s}\; a \diamond c \diamond b \diamond d \in \mathsf{comb}(a \diamond b, c \diamond d)}{acbd \in \mathrm{L}(G).}\; \mathsf{lock}(a \diamond c \diamond b \diamond d) = acbd$$

For the first derivation to be possible, the diamonds in $a \diamond b$ and $c \diamond d$ are still required; otherwise, we would have $\mathsf{comb}(\{ab\}, \{cd\}) = \{abcd, cdab\}$, which does not contain $acbd$. Keeping the diamonds in place until the end of the derivation process is therefore essential.

The following result comes "for free":

Proposition 2 PMCFL $\subset$ IDLPMCFL.

We finally give a simple example of how this can be used to implement a very simple grammar. Suppose that we want to encode a small subset of Latin that contains sentences composed of a final verb and an optional subject. This subject is a noun phrase, i.e. a noun to which an arbitrary number of optional adjectives may be attached. The following IDL-CFG describes exactly this:

$$\begin{gathered} NP \to V \to S : np, v \mapsto np \cdot v \\ V \to S : v \mapsto v \end{gathered}$$

$$N \to NP : n \mapsto n$$
$$NP \to A \to NP : np, a \mapsto || (np, a) .$$

Remark that while we used the || operator for building a new NP from an NP and an adjective (meaning that the adjective is allowed to appear either before, within or after the NP it is appended to), we resorted to simple concatenation for building a sentence from an NP and a verb, as we want the verb to appear at the end of the sentence, after the subject NP.

2.4 Expressivity

We shall now investigate the expressivity of IDL-(PM)CFGs and try to locate the corresponding class of languages within the hierarchy of polynomial languages. Recall the following series of inclusions

$$\mathrm{CFL} \subsetneq \mathrm{TAL} \subsetneq \mathrm{PMCFL} \subsetneq \mathrm{PRCL} = \mathrm{P}$$

where

- CFL is the class of context-free languages;
- TAL is the class of tree-adjoining languages (Vijayashanker and Joshi 1988);
- PMCFL is the class of parallel multiple context-free languages;
- PRCL is the class of positive range concatenation languages (Boullier 1998);
- P is the class of languages recognizable in polynomial time.

The important equality PRCL = P is proved in Boullier (1998).

The main contribution of this subsection is a proof that IDL-PMCFGs can be located strictly above PRCGs in the hierarchy (Theorem 1). We also show that IDL-CFGs are strictly more expressive than TAGs and not more expressive than PMCFGs and suggest the question of whether IDLCFL $\subset$ PMCFL as a natural generalization of a recently solved classification problem.

We first observe that IDL-CFG allows us to define in a very compact manner the nMIX language family:

Proposition 3 *For all $n \in \mathbb{N}^+$, the n*MIX *language defined as*

$$n\mathrm{MIX} = \left\{x \in \{a_1, \ldots, a_n\}^* \mid |x|_{a_1} = \cdots = |x|_{a_n}\right\}$$

is in IDLCFL.

Proof Let $n \in \mathbb{N}^+$. The following IDL-CFG

$$S : \varepsilon$$
$$S \to S : s \mapsto || (s, a_1, \ldots, a_n)$$

defines nMIX. □

The position of the nMIX languages within the hierarchy has been intensively studied within the last decade. Language 2MIX is context-free. For $n \geq 3$, the problem turns out to be much more difficult. The original MIX language (or Bach language), i.e. 3MIX, has been proven a PMCFL by Salvati (2015). Makoto and Salvati (2012) also proved that MIX is not a tree-adjoining language. Together with the fact that IDL-CFGs generate all nMIX languages, this results provides us with the following corollary:

Proposition 4 IDLCFL $\not\subset$ TAL.

For many years, no general classification results were available for $n > 3$. Only very recently did Ho (2018) prove that for all n, the word problem of $\mathbb{Z}^n$ is in PMCFL. Since the word problem of $\mathbb{Z}^n$ and nMIX are rationally equivalent (Salvati 2015), this yields the inclusion of the whole nMIX family within PMCFL.

Proving that IDLCFL $\subset$ PMCFL would be an even stronger result, given that nMIX $\in$ IDLCFL for all n; the inclusion might even appear likely in the light of Ho's proof. Nevertheless, the amount of work needed to address the "specific" case of nMIX languages suggests that this will be anything but easy, and we will leave this for future work.

On the other hand, it is clear that IDL-CFG does not contain PMCFG.

Proposition 5 *Language* L_{3n} *above is not in* IDLCFL.

Proof By contradiction, let G be an IDL-CFG matching L. Consider the context-free grammar G' that is obtained from G by replacing every IDL expression e over some alphabet Σ' in the right-hand side of a rule by a string $s(e)$ defined inductively as follows:

$$\begin{aligned}
s(a) &= a && \forall a \in \Sigma' \cup \{\underline{\varepsilon}\} \\
s\left(e \cdot e'\right) &= s(e) \cdot s\left(e'\right) && \\
s(\times(e)) &= s(e) && \\
s(\vee(e_1, \ldots, e_n)) &= \vee(s(e_1) \cdot \cdots \cdot s(e_n)) && \forall n \in \mathbb{N} \\
s(||(e_1, \ldots, e_n)) &= s(e_1) \cdot \cdots \cdot s(e_n) && \forall n \in \mathbb{N}.
\end{aligned}$$

Note that an equivalent CFG grammar in a more canonical form can be easily obtained by removing the disjunction nodes in exchange of an increase in the number of rules. Now, it is straightforward that the language L' generated by G' is a subset of L. Moreover, for any string w generated by G, there exists a string $w' \in \mathrm{L}' \subset \mathrm{L}$ such that $|w| = \left|w'\right|$, and such that w' is obtained by applying the same rules in G' that were used to produce w in G. By construction, w' is a permutation of w. Let $x \in \mathrm{L}$, and $n = |x|$. By definition of L, x is the only word of length n in L. As a consequence, $x \in \mathrm{L}'$. This means that $\mathrm{L}' = \mathrm{L}$ and that the CFG grammar G' recognizes L, which is impossible since L is not context-free. ◻

Again, this results in a classification result:

Proposition 6 PMCFL $\not\subset$ IDLCFL.

We have this corollary:

Proposition 7 IDLCFL $\subsetneq$ IDLPMCFL.

Proof By Propositions 2 and 6. □

An essential result is that unless P $=$ NP, IDL-PMCFGs can define some non-polynomial languages. This is in line with Kirman and Salvati (2013)'s findings that even classes of grammars that are "close to [...] mildly context sensitive" may have NP-hard membership problems as soon as commutation is allowed. In the case of IDL-PMCFG, we will prove this in three steps. First, we will recall the definition of the NP-complete problem 3SAT and suggest a polynomial encoding of it on a simple finite alphabet. Second, we will construct an IDL-PMCFG grammar that recognizes the language of satisfiable 3-CNF formulae in the previous encoding. A final step will then lead us to the result.

The 3SAT problem is one of Karp (1972)'s 21 NP-complete problems. It asks for determining whether a finite boolean formula on a potentially infinite set of variables $\{x_n\}_{n\in\mathbb{N}}$, input in conjunctive normal form (CNF) with at most three literals per clause, is satisfiable. Consider for example the 3-CNF formulae

$$\begin{aligned} f_1 &= (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \\ f_2 &= (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3) \wedge \neg x_1 \wedge \neg x_3. \end{aligned}$$

Formula f_1 is satisfiable because the valuation $x_1 \mapsto \top, x_2 \mapsto \bot, x_3 \mapsto \top$ results in the formula reducing to $\top$. Formula f_2 is not satisfiable: the last two unary clauses impose $x_1 \mapsto \bot, x_3 \mapsto \bot$, but then the first clause requires $x_2 \mapsto \bot$ to be satisfied whereas the second one needs $x_2 \mapsto \top$, a contradiction.

The size of 3-CNFs is measured by the number of their clauses, without regard to the number of variables. In the two instances above, this gives $|f_1| = 3$, $|f_2| = 4$.

So far, our description of 3-CNF formulae, unlike the grammars we study in this paper, used an infinite alphabet to encode variables. We now introduce an encoding of 3-CNF logical formulae on an finite alphabet.

Definition 9 Let $\Sigma = \{\texttt{[}, \texttt{]}, \texttt{(}, \texttt{)}, \texttt{1}, \texttt{!}\}$. Let $V = (x_n)_{n\in\mathbb{N}^+}$ the set of variables and $SV = V \cup \neg V$ the set of optionally negated variables. We define

– A mapping $\nu : V \to \Sigma^*$ encoding variables as the unary representation of their index:

$$\forall n \in \mathbb{N},\ \nu(x_n) = \texttt{1}^n;$$

– A mapping $\mu : SV \to \Sigma^*$ encoding optionally negated variables by appending the character ! in front of negated variables:

$$\forall v \in V, \begin{cases} \mu(v) = \nu(v) \\ \mu(\neg v) = \texttt{!} \cdot \nu(v) \end{cases};$$

– A mapping $\pi : SV^3 \to \Sigma^*$ encoding ternary clauses in the following way:

$$\forall (u, v, w) \in SV^3,\ \pi (u \vee v \vee w) = (\ \cdot \mu (u) \cdot\ ) \cdot (\ \cdot \mu (v) \cdot\ ) \cdot (\ \cdot \mu (w) \cdot\ )\ ;$$

– A mapping $\tau : \bigcup_{n\in\mathbb{N}} \left(SV^3\right)^n \to \Sigma^*$ encoding 3-CNF formulae as follows:

$$\begin{aligned}&\forall n \in \mathbb{N}, \forall (C_1, \ldots, C_n) \in \left(SV^3\right)^n,\\ &\tau (C_1 \wedge \cdots \wedge C_n) = [\ \cdot \pi (C_1) \cdot\ ] \cdot \cdots \cdot [\ \cdot \pi (C_n) \cdot\ ]\ .\end{aligned}$$

Mappings ν, μ, π and τ are clearly bijective.

The above encoding is only applicable to formulae in 3-CNF where every clause contains exactly three literals. A straightforward observation makes this restriction largely irrelevant and will simplify the discussion later on:

Proposition 8 *Let f be a logical formula in 3-CNF. There exists another logical formula $\hat{f}$ in 3-CNF such that:*

1. *Formulae f and $\hat{f}$ are equivalent and $\left|\hat{f}\right| = |f|$;*
2. *All clauses in $\hat{f}$ have exactly three literals;*
3. *The set $\hat{W}$ of variables used in $\hat{f}$ is equal to $(x_n)_{n\in[\![1,N]\!]}$ for some $N \in \mathbb{N}$ such that $N \le 3\,|f|$.*

Moreover, for all such f, a formula $\hat{f}$ matching the three above conditions can computed from f in time $\mathcal{O}(|f|)$.

Proof Let f be a logical formula in 3-CNF and W the set of its variables. We derive $\hat{f}$ from f as follows:

1. Rename the variables in f to produce a new formula g with variable set X such that $X = (x_n)_{n\in[\![1,|W|]\!]}$. One convenient way to achieve this is to process the formula from left to right, keeping in mind the index of the smallest currently unused variable in the new (partial) formula, as well as the correspondance between variables in f that have already been renamed in g. This is done in time linear in $|f|$ and would e.g. convert $f_3 = (x_3 \vee x_1 \vee x_{17}) \wedge (x_4 \vee \neg x_3 \vee \neg x_{16})$ into $g_3 = (x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee \neg x_1 \vee \neg x_5)$.
2. Now, for every clause that has only one (resp. two) literals, duplicate (resp. triplicate) the first literal of the clause. This can be done in linear time scanning the input from left to right. For instance, formula f_2 would be converted into $g_2 = (x_1 \vee \neg x_2 \vee x_1) \wedge (x_2 \vee \neg x_3 \vee x_2) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_1) \wedge (\neg x_3 \vee \neg x_3 \vee \neg x_3)$.

Since the total number of variables in a 3-CNF formula cannot exceed 3 times the number of clauses, the resulting formula $\hat{f}$ clearly satisfies the above conditions. □

Applying the operations described in Proposition 8 followed by the mapping τ described in Definition 9 leads to the following encoding of formulae f_1 and f_2:

$$\tau\left(\hat{f}_1\right) = \texttt{[(1)(11)(!111)][(!11)(!111)(!11)][(!1)(11)(111)]}$$
$$\tau\left(\hat{f}_2\right) = \texttt{[(1)(!11)(1)][(11)(!111)(11)][(!1)(!1)(!1)]}$$
$$\times\ \texttt{[(!111)(!111)(!111)]}.$$

The following lemma provides the key argument:

Lemma 3 *Let* 3SATL *be the language of satisfiable 3-CNF formulae. There exists some IDL-PMCF grammar G such that, for all 3-CNF formula f, $f \in$* 3SATL *iff $\tau\left(\hat{f}\right) \in \mathrm{L}(G)$.*

Proof We build an IDL-PMCF grammar G that recognizes satisfiable 3-CNF formulae encoded as in Definition 9. First, we define variables (of category V and arity 1) as sequences of `1`s:

$$V : \langle \texttt{1} \rangle$$
$$V \to V : v \mapsto \langle v[0] \cdot \texttt{1} \rangle .$$

We proceed by defining literals (of category L, arity 1) as variables preceded by the optional negation symbol `!`:

$$V \to L : v \mapsto \left\langle \vee\left(\underline{\varepsilon}, \texttt{!}\right) \cdot v[0] \right\rangle .$$

A satisfiable formula and the valuation satisfying it are produced in parallel through a number of *double-steps*, each of them consisting of:

1. A *selection step* where a new variable $x_{i'} := x_{i+1}$ is selected and its boolean value v_i in the valuation is chosen;
2. An *insertion step* where an arbitrary number of ternary clauses containing x_i (if $v = \top$) or $\neg x_i$ (if $v = \bot$) is added at arbitrary positions in the already produced formula.

Each double-step uses a category F of arity 3 that stores the current formula f_i as well as the current litteral $\sigma_i x_i$ (where $\sigma_i \in \{\varepsilon, \texttt{!}\}$ and $\sigma_i = \varepsilon \Leftrightarrow v_i = \top$) as $\langle f_i, x_i, \sigma_i \rangle$.[4]

The selection step retrieves the next variable and chooses its value by

$$F \to F : f \mapsto \left\langle f[0], f[1] \cdot \texttt{1}, \underline{\varepsilon} \right\rangle$$
$$F \to F : f \mapsto \langle f[0], f[1] \cdot \texttt{1}, \texttt{!} \rangle .$$

The insertion step adds arbitrary clauses containing the current litteral to the current formula:

$$L \to L \to F \to F : v, w, f \mapsto \langle ||(f[0], \times(\texttt{[} \cdot ||(\times(\texttt{(} \cdot v[0] \cdot \texttt{)}),$$
$$\times(\texttt{(} \cdot w[0] \cdot \texttt{)}), \times(\texttt{(} \cdot f[2] \cdot f[1] \cdot \texttt{)})) \cdot \texttt{]})), f[1], f[2] \rangle .$$

4 We thank Peter Ljunglöf for suggesting this simpler expression of the grammar.

This rule can be described informally as follows: interleave into the current formula a (locked) clause consisting of three interleaved (locked) literals, the first two of which are arbitrary while the third one is equal to the current litteral; literals are enclosed in parentheses while clauses are enclosed in square brackets.

Finally, we have to indicate that the start category can be produced from any F and that the empty formula is satisfiable (with $\mathtt{1}$ as first variable to consider):

$$F : \left\langle \underline{\varepsilon}, \mathtt{1}, \underline{\varepsilon} \right\rangle$$
$$F : \left\langle \underline{\varepsilon}, \mathtt{1}, \mathtt{!} \right\rangle$$
$$F \to S : f \mapsto \langle f[0] \rangle .$$

Let f be a 3-CNF formula such that $f \in$ 3SATL. As f and $\hat{f}$ are equivalent, $\hat{f} \in$ 3SATL. Let N such that the set $\hat{W}$ of variables of $\hat{f}$ is equal to $(x_n)_{n \in [\![1,N]\!]}$. Let v be a valuation of $\hat{W}$ satisfying $\hat{f} =: \left(C_1, \ldots, C_{|f|}\right)$. Let $(E_i)_{i \in [\![1,N]\!]} \in \mathscr{P}\left([\![1, |f|]\!]\right)^N$ such that $\sqcup_{i=1}^{N} E_i = [\![1, |f|]\!]$ and for all $i \in [\![1, N]\!]$, $n \in [\![1, |f|]\!]$,

$$n \in E_i \Rightarrow (v(x_i) = \top \wedge x_i \in C_n) \vee (v(x_i) = \bot \wedge \neg x_i \in C_n) .$$

In other words, E_i is a set of clauses such that the current value of x_i in v makes all clauses in the set reduce to true. Let $\kappa(\top) = \underline{\varepsilon}$ and $\kappa(\bot) = \mathtt{!}$. This decomposition necessarily exists since $\hat{f}$ is satisfiable, but it is in general not unique. The string $\tau\left(\hat{f}\right)$ is recognized by G using the following derivations for $i = 1$ up to N, starting with $\left\langle \underline{\varepsilon}, \mathtt{1}, \kappa(v(x_1)) \right\rangle$ of category F:

- Consider the available item $\phi = \left\langle f, \mathtt{1}^i, \kappa(v(x_i)) \right\rangle$ of category F;
- For all $n \in E_i$:
  - Without loss of generality, suppose $v(x_i) = \top$ and therefore $\kappa(v(x_i)) = \underline{\varepsilon}$,
  - Up to reordering of i, j and k, $C_n = x_i \vee \sigma_j x_j \vee \sigma_k x_k$,
  - Produce two literals of category L containing $\sigma_j x_j$ and $\sigma_k x_k$ respectively,
  - Use them along with ϕ to produce $\phi' = \left\langle f', \mathtt{1}^i, \underline{\varepsilon} \right\rangle$ where f' is equal to f up to a clause

    $$\mathtt{[\ (1}^i\mathtt{)\ (}\kappa\left(\sigma_j\right)\mathtt{1}^j\mathtt{)\ (}\kappa\left(\sigma_k\right)\mathtt{1}^k\mathtt{)\ ]}$$

    that has been added at a position compatible with the final reordering in $\tau\left(\hat{f}\right)$,
  - Do $\phi := \phi'$, $f := f'$;
- If $i < N$, produce an item $\phi := \left\langle f, \mathtt{1}^{i+1}, v(x_{i+1}) \right\rangle$ of category F;
- Else, $i = N$: produce $\langle f \rangle$ of category S; terminate.

Conversely, let f be a 3-CNF formula such that $\tau\left(\hat{f}\right) \in \mathrm{L}(G)$. As f and $\hat{f}$ are equivalent, it suffices to prove that $\hat{f} \in$ 3SATL. Now, it is straightforward to see that subsets E_i of $[\![1, |f|]\!]$ verifying the same properties as in the first half of the proof can be constructed by considering the clauses added in f at iteration i. The existence of these subsets guarantees that $\hat{f} \in$ 3SATL, which concludes the proof. □

Our main classification result follows:

Theorem 1 *Unless* P = NP, IDLPMCFL $\not\subset$ PRCL = P.

Proof By contradiction, suppose that IDLPMCFL $\subset$ P. We will now prove that 3SATL $\in$ P.

Let G be the grammar defined in Lemma 3. Thanks to our hypothesis that IDLPMCFL $\subset$ P, the language $\mathrm{L}(G)$ recognized by G is in P. Let T be a (deterministic) Turing machine that recognizes $\mathrm{L}(G)$ in polynomial time. Consider the following procedure POLY3SAT:

Algorithm 1

```
1: procedure POLY3SAT($f$: 3-CNF formula)
2:     $g := \hat{f}$
3:     $h := \tau(g)$
4:     return $T(h)$
```

First, this procedure runs in polynomial time in the size $|f|$ of the input:

1. Computing $\hat{f}$ takes time $\mathcal{O}(|f|)$ according to Proposition 8, and $\left|\hat{f}\right| = |f|$.
2. Computing $\tau(g)$ is also clearly linear in the size $|g| = |f|$ of its input. The size of $\tau(g) \in \Sigma^*$ is given by its length. The set W of variables appearing in g is included in $(x_i)_{i\in[\![1,3|g|]\!]}$ according to Proposition 8. Therefore, $|\nu(w)| \leq 3|g| = 3|f|$ for all $w \in W$. Following Definition 9, we get

$$|\tau(g)| \leq |g|(8 + 3 \times (1 + 3|f|)) = |f|(8 + 3 \times (1 + 3|f|)) = \mathcal{O}\left(|f|^2\right);$$

3. Finally, computing $T(h)$ is by assumption a $\mathcal{O}(|h|^\alpha)$ for some $\alpha \in \mathbb{N}^+$. As $|h| = |\tau(g)| = \mathcal{O}\left(|f|^2\right)$, computing $T(h)$ is a $\mathcal{O}\left(|f|^{2\alpha}\right)$.

Second, it recognizes 3SATL. This is a direct consequence of Lemma 3: $f \in$ 3SATL iff $h = \tau\left(\hat{f}\right) \in \mathrm{L}(G) = \mathrm{L}(T)$, iff POLY3SAT returns true when applied to f.

We have proved that POLY3SAT recognizes 3SATL in polynomial time. Hence, 3SATL $\in$ P. As 3SAT is NP-complete, this yields P = NP. □

This theorem, combined with results obtained by Ljunglöf (2005), admits a useful corollary:

Theorem 2 *Unless* P = NP, PMCFL $\subsetneq$ IDLPMCFL.

Proof Ljunglöf (2005) proves that PMCFL $\subsetneq$ PRCL = P. According to Theorem 1, unless P = NP, we have IDLPCMFL $\not\subset$ P. Clearly, PMCFL $\subset$ IDLPMCFL, so unless P = NP, PMCFL $\subsetneq$ IDLPMCFL. □

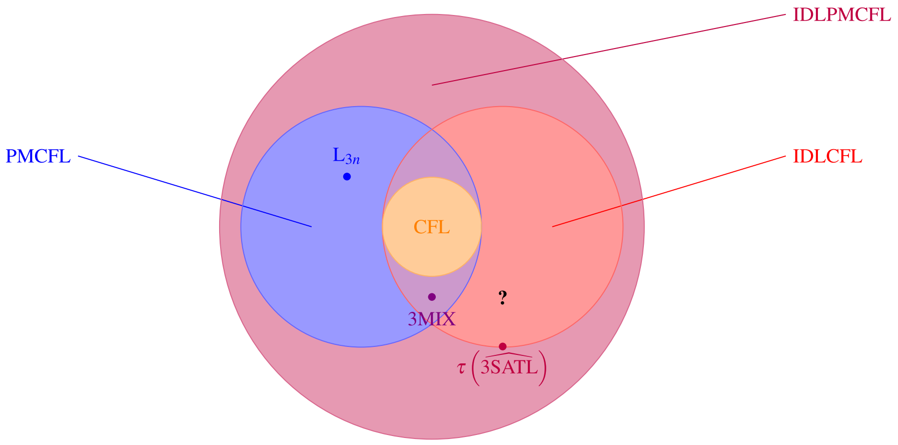


Fig. 1 PMCFL, IDLCFL and IDLPMCFL, unless P = NP

Our results are summarized on Fig. 1. The following questions remain open:

1. Is IDLCFL $\subset$ PMCFL? We have suggested above that the answer is likely to be positive. This is displayed on Fig. 1 by the interrogation mark and the fact that language $\tau\left(\widehat{3SATL}\right) \in$ IDLPMCFL $\subset$ PMCFL is placed at the border between IDLCFL$\setminus$ (PMCFL $\cup$ IDLCFL) and IDLCFL $\cap$ (IDLPMCFL$\setminus$PMCFL).
2. Is IDLPMCFL $\subset$ CSL, i.e., are all IDLPMCFLs recognized by some linear bounded automaton?

Independently from the answers to the previous questions, it is already clear that the presence in a CFL-based formalism of all three $||$, $\cdot$ and $\times$ operators as well as of tuples of size at least 2 and copying, is sufficient to leave the realm of P. As noted in Sect. 1.2, interleaving, linear constraints, locking, records and copying are reasonable requirements for a grammatical formalism designed to describe the syntax of free word order languages in general, and of Classical Latin in particular. This, of course, does not mean that Classical Latin itself would be non-polynomial, since the reduction presented is not linguistically relevant, and involves copying which Latin does not require. It simply means that a grammatical formalism for free word order languages containing the features above leads to worst-case non-polynomial scenarios which might not necessarily be linguistically relevant.

3 COMPĀ: A Programming Language for Describing Free Word Order Syntax

3.1 Grammatical Framework and COMPĀ

Grammatical Framework (GF) (Ranta 2004), developed by Ranta et al. since 1998, is a special-purpose programming language aimed at writing grammars of natural languages. Practically, GF serves as the natural-language counterpart of tools such as YACC (Johnson et al. 1975) or Menhir (Pottier and Régis-Gianas 2016) for programming languages. From a logical point of view, Grammatical Framework is a logical framework relying on Martin-Löf type theory (Martin-Löf and Sambin 1984). A functional programming language, GF also has a large support for modularity and enforces conventions and standards simplifying the development of multilingual applications. Its community actively contributes to the Resource Grammar Library (Ranta et al. 2009), that unites 'concrete' wide-coverage grammars for over 30 individual languages around a common 'abstract' grammar. Over the course of the last 20 years, GF, which remains fully open-source, has been used in several experimental as well as industrial contexts, for applications ranging from morphological generation to natural language transcription of formal (mathematical, proof, technical) language, from multilingual translation of 'controlled language' to language learning tools.

This chapter describes COMPĀ, a GF-like programming language tailored to encode the grammatical syntax of free word order languages. Though it has been primarily conceived to model and study the syntax of the Latin language, its design as well as the description we will give are both language-agnostic. The name COMPĀ stands for *COMPĀgēs Grammaticālis Latīna*, which means 'Latin Grammatical Framework' in Latin.

The syntax and semantics of COMPĀ are largely borrowed from standard GF: it is a functional programming language in Haskell-style manipulating sets of words/terminals, and providing records and tables over finite types, as well as finite lambda functions (Ranta 2011). As an experimental language focussing on the syntactic description of individual languages, COMPĀ does not implement structures and operators mainly directed at handling morphology, semantics and multilingualism or providing additional modularity, such as abstract grammars, dependent types, token-level gluing or general lambda functions. More precisely, COMPĀ's extends a subset of GF so-called *context-free GF* (Ljunglöf 2004). In turn, it provides 3 operators absent from standard GF, viz. the interleave ($||$), disjunction ($\vee$) and lock ($\times$) operators. While standard GF compiles (mostly for parsing purposes) into Angelov et al. (2009)'s PMCFG-equivalent PGF, COMPĀ can be transcompiled into IDL-PMCFG.

In this section, we will focus on aspects of COMPĀ's design that differ from standard GF, and show how it can be used as an efficient front-end for writing practical grammars of free word order languages. For a detailed presentation of the syntax of standard GF, the reader can refer to the GF reference manual (Ranta 2011).

3.2 Operations and Types

3.2.1 Data Types

As a language designed to model free word order languages, COMPĀ relies one fundamental data type Set, the *type of sets of token lists* (short: 'sets'), that replaces the standard GF Str. The basic operators described below take as input, and return, only data of type Set.

Besides the fundamental type Set, each grammar may define an arbitrary number of *parameter types*. These finite types are often used to encode specific grammatical features (e.g. case, number or gender).

Record types can be built from a list of (distinct) identifier names and a list of types, each of which might be either the type Set or any finite type. Records store structured information and allow for an accurate representation of grammatical constituents (storing some of their features as well of one or several sets that represent their linearization).

Tables are finite functions that map every value of a finite type to a value of some (unique) other type.

Given a set of finite (i.e. enumerated) types Π and the set of admissible string identifiers S, the syntax of admissible types is formally defined as follows (Fig. 2):

$$\begin{array}{lr}
\tilde{\tau} := \mathsf{Set} & \\
\quad \mid \{i_1 : \tilde{\tau}_1; \ldots; i_k : \tilde{\tau}_k\} & \forall k \in \mathbb{N}, \forall \{i_1, \ldots, i_k\} \in \mathscr{P}_k(\mathscr{S}) \\
\quad \mid \pi \Rightarrow \tilde{\tau} & \forall \pi \in \Pi \\
\quad \mid \pi & \forall \pi \in \Pi
\end{array}$$

Fig. 2 COMPĀ's types

3.2.2 Operations on Sets

Sets are introduced by means of the standard syntax for strings. Thus, in COMPĀ, the expression `consul` does not represent the singleton token list [`consul`] as in GF, but it instead stands for the singleton set {[`consul`]}. Similarly, COMPĀ's `[]` does not stand as in GF for the empty list of tokens (the *empty string*), but for the singleton containing the empty list of tokens. Another more practical way to put it is to see this set as the set of possible phrases that can be derived from the expression `consul`: there is only one, containing one word, the word `consul`, hence the singleton set above. Note that the empty set of strings has its own syntax, `variants {}`, that is also borrowed from standard GF.

COMPĀ define four basic operations on sets, that are the exact counterparts of those defined in Nederhof and al.'s IDL expressions formalism (Fig. 3):

Name	**COMPĀ**	**Nederhof and al.**	**Syntax**	**Associative**	**Priority**
Concatenation	`++`	$\cdot$	`a ++ b`	Left and right	3
Interleave	`||`	$\|\|$	`a || b`	Left and right	2
Disjunction	`\/`	$\vee$	`a \/ b`	Left and right	1
Lock	`‘`	$\times$	`‘a`	Right	4

Fig. 3 Operations on sets in COMPĀ

3.3 Syntax

3.3.1 Structure of Programs

Each COMPĀ program, called a *grammar*, is enclosed in a file, with each COMPĀ file defining exactly one such grammar. Standard extension for COMPĀ files is `.cp`. The syntax of programs is as follows (note that all whitespace and line breaks are ignored) (Fig. 4):

The identifier following the `concrete` keyword is an (arbitrary) name. We will now go through each of the four sections of the grammar and consider their individual syntaxes.

Fig. 4 Syntax of COMPĀ identifiers and grammars

$$\begin{aligned}
&\textit{ident} := [\texttt{a}-\texttt{zA}-\texttt{Z\_}]\,[\texttt{0}-\texttt{9a}-\texttt{zA}-\texttt{Z\_'}]*\\
&\textit{prog} := \texttt{concrete}\ \textit{ident} = \{\\
&\qquad [\texttt{includecc}\,[\textit{include}_1;[\textit{include}_2;\ldots]]]\\
&\qquad [\texttt{param}\,[\textit{param}_1;[\textit{param}_2;\ldots]]]\\
&\qquad [\texttt{lincat}\,[\textit{lincat}_1;[\textit{lincat}_2;\ldots]]]\\
&\qquad [\texttt{lin}\,[\textit{lin}_1;[\textit{lin}_2;\ldots]]]\\
&\}
\end{aligned}$$

3.3.2 Including GF Lexica

The first section is used to import already extant standard GF lexica into COMPĀ. Its syntax is extremely simple (Fig. 5):

where *filename* is the name of a concrete GF file functioning as a lexicon. When an *include* is read, the corresponding GF file is retrieved and all words it defines are automatically extracted. The *include* section thus provides some compatibility with standard GF as well as a support (through the use of GF itself) for efficient morphological analysis.

Fig. 5 Syntax of *include*

$$\textit{include} := \texttt{"}\textit{filename}\texttt{"}$$

3.3.3 Parameters

Parameters are declared as follows (Fig. 6):

In the above description, $\textit{ident}_0$ is the name of the new finite type, and $(\textit{ident}_k)_{k\geq 1}$ its values. Both type and value parameter identifiers must be unique throughout the whole grammar, and are usually (but not necessarily) capitalized.

Fig. 6 Syntax of *param*

$$\textit{param} := \textit{ident}_0 = \textit{ident}_1\ [\,|\ \textit{ident}_2\ [\,|\ \textit{ident}_3 \ldots]]$$

3.3.4 Categories

Categories are introduced in the *lincat* section according to the syntax below, where *paramType* is any parameter type defined in the *param* section (Fig. 7):

$$\begin{aligned}
\mathit{lincat} &:= \mathit{ident} \texttt{ = } \mathit{type} \\
\mathit{type} &:= \texttt{Set} \\
&\mid \mathit{paramType} \\
&\mid \texttt{\{}\,[\mathit{ident}_1 \texttt{:} \mathit{type}_1\, [\texttt{;} \mathit{ident}_2 \texttt{:} \mathit{type}_2\, [\texttt{;} \mathit{ident}_3 \texttt{:} \mathit{type}_3 \ldots]]]\,[\texttt{;}]\,\texttt{\}} \\
&\mid \mathit{paramType} \texttt{ => } \mathit{type}
\end{aligned}$$

Fig. 7 Syntax of *lincat*

3.3.5 Linearization Functions

The *lin* section collects the functional rules that are the heart of any GF or COMPĀ grammar. Each linearization rule describes a way to combine several arguments (of given input categories) into a new item (of a given output category). Unlike GF, which separates the type-annotated declaration of linearization functions in abstract syntax files from the non-type-annotated definition of linearization functions in concrete syntax files, COMPĀ uses only a single (concrete) syntax, which is directly annotated with types. COMPĀ includes a complete type-checker.

Let us first formally describe the syntax of linearization functions. In this figure, the non-terminals *paramType* and *paramValue* match parameter types and values introduced in the *param* section, whereas the non-terminal *licatName* matches the name of any category defined in the *lincat* section. In the definition of *lin*, ident_0 and $\mathit{lincatName}_0$ are respectively the name of the linearization function and its output category, while

$$(\mathit{ident}_k, \mathit{lincatName}_k)_{k\geq 1}$$

are the names and categories of the function's arguments (Fig. 8).

$$\begin{aligned}
\mathit{lin} &:= \mathit{ident}_0\, [\texttt{(}\mathit{ident}_1 \texttt{:} \mathit{lincatName}_1\texttt{)}\, [\texttt{(}\mathit{ident}_2 \texttt{:} \mathit{lincatName}_2\texttt{)} \ldots]] \texttt{:} \mathit{lincatName}_0 \texttt{ = } \mathit{expr} \\
\mathit{expr} &:= \texttt{[]} \\
&\mid \texttt{"}\mathit{string}\texttt{"} \\
&\mid \mathit{ident} \\
&\mid \texttt{(}\mathit{expr}\texttt{)} \\
&\mid \texttt{`}\mathit{expr} \\
&\mid \texttt{not } \mathit{expr} \\
&\mid \mathit{expr}_1 \texttt{!} \mathit{expr}_2 \\
&\mid \mathit{expr}\texttt{.}\mathit{ident} \\
&\mid \mathit{expr}_1 \texttt{ ++ } \mathit{expr}_2 \\
&\mid \mathit{expr}_1 \texttt{ || } \mathit{expr}_2 \\
&\mid \mathit{expr}_1 \texttt{ \textbackslash/ } \mathit{expr}_2 \\
&\mid \texttt{table \{}\mathit{paramValue}_1 \texttt{ => } \mathit{expr}_1\, [\texttt{;}\mathit{paramValue}_2 \texttt{ => } \mathit{expr}_2\, [\texttt{;}\mathit{paramValue}_3 \texttt{ => } \mathit{expr}_3 \ldots]]\,[\texttt{;}]\,\texttt{\}} \\
&\mid \texttt{table } \mathit{paramType}\ \texttt{[}\mathit{expr}_1\, [\texttt{;}\mathit{expr}_2\, [\texttt{;}\mathit{expr}_3 \ldots]]\,[\texttt{;}]\,\texttt{]} \\
&\mid \texttt{\textbackslash\textbackslash}\mathit{ident}\texttt{:}\mathit{paramType} \texttt{ => } \mathit{expr} \\
&\mid \texttt{\{}\,[\mathit{ident}_1 \texttt{ = } \mathit{expr}_1\, [\texttt{;}\mathit{ident}_2 \texttt{ = } \mathit{expr}_2\, [\texttt{;}\mathit{ident}_3 \texttt{ = } \mathit{expr}_3 \ldots]]]\,[\texttt{;}]\,\texttt{\}} \\
&\mid \texttt{for } \mathit{ident}\texttt{:}\mathit{paramType} \texttt{ do } \mathit{expr}
\end{aligned}$$

Fig. 8 Syntax of *lin*

3.3.6 Iterating over Finite Types with `for`

To handle those cases where similar rules must be constructed for all possible values of a given parameter, a loop structure absent from standard GF is proposed. This structure is available in COMPĀ through a `for-do` construction.

Suppose that verb category V has type $\{s : \mathsf{Tense} \Rightarrow \mathsf{Set}\}$ where Tense is a finite type enumerating the available tenses in the language, and that we want to write a rule that takes a verb of category V and produces a conjugated verb of category *ConjugV* and type $\{s : \mathsf{Set}; \textit{tense} : \mathsf{Tense}\}$ that stores a conjugated verb and keeps trace of its tense. This can be achieved in COMPĀ like this:

```
conjugateVerb (v : V) : ConjugV
 = for t : Tense do { s = v.s ! t; tense = t };
```

When translated into low-level IDL-PMCFG, this results into several parallel rules being constructed, one for each available value of the bound variable. This is especially useful when a parameter (e.g. a verb tense or mode) provides different linearizations without playing any part in the syntactic structure itself, or when another parameter (e.g. number) can be arbitrarily chosen at some syntactic level before being propagated downwards into the tree.

3.4 The COMPĀ (Trans)Compiler

Just as standard GF must be compiled into low-level PMCFG for parsing purposes, the COMPĀ language is used as a grammar description front-end that has to be translated into IDL-PMCFG before parsing.

Using OCaml, we implemented a lightweight transcompiler that type-checks and converts a COMPĀ grammar into an equivalent IDL-PMCFG grammar. The essential conversion step employs finite function resolution techniques similar to those presented by Ljunglöf (2004): tables and parameter fields are replaced by new fields and categories, and new rules are finally created between new categories.

The compiler's source code can be found in the corresponding GitHub repository.[5]

4 A Parsing Algorithm for COMPĀ

In this section, we present a parsing algorithm for IDL-PMCF grammars and provide an analysis of its complexity. This algorithm, for which we also provide a complete OCaml implementation, is inspired by the works of Ljunglöf (2012) and Angelov (2009) on GF parsing, while building on techniques introduced by Nederhof and Satta (2004) to parse IDL expressions. We extend Nederhof and Satta's graph-based finite state approach, enriching it by decorating active nodes by sets of word positions.

[5] See https://github.com/Streichholzschaechtelchen/l2ud.

4.1 Parsing COMPĀ's IDL Expressions

Nederhof and Satta (2004) present a parsing algorithm for IDL expressions relying on left-to-right scanning of the input and a representation of the current parsing state as a *cut* (a set of nodes verifying certain properties, that does not necessarily match the traditional graph-theoretical definition of a cut—see below) within a so-called IDL graph. Each IDL expression is compiled to a single IDL graph. Transitions from one state/cut to another state/cut are encoded in the IDL graph; edges are labelled with words that must be read to transition from one cut to another. The input is parsed successfully if and only if the final state is reached after all characters have been read.

Unlike in the original publication, where IDL expressions were used as autonomous regular expressions rather than within a grammar, the edges of COMPĀ's IDL expressions may be annotated both by terminals, i.e. words, and by (nonterminal, index) pairs. The latter labelling corresponds to the case where we want to match a field of one of the arguments of the current rule.

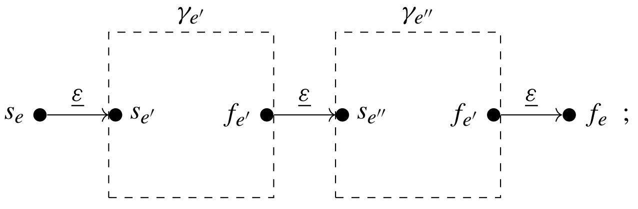
Let us now define the IDL graph associated to a given IDL expression. Note that this definition, though closely following along the lines of Nederhof and Satta's contribution, does not encode the lock operator in the same way. This different encoding has been found more practical for parsing of full IDL-PMCF grammars, as will be overt when we will discuss our algorithm.

Definition 10 (IDL graph) Let $G = (N, \delta, \Sigma, F, P, S)$ be an IDL-PMCFG.

Let $\left(f, A_1, \ldots, A_q, A\right) \in P$. Let e be an IDL expression over $\Sigma' = \Sigma \cup \left\{X_{ij} \mid i \leq a\left(f\right), j \leq d_i\left(f\right)\right\}$.

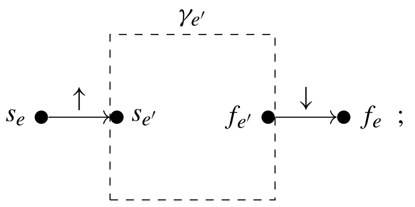
The IDL graph γ_e associated with e is defined by induction as follows:

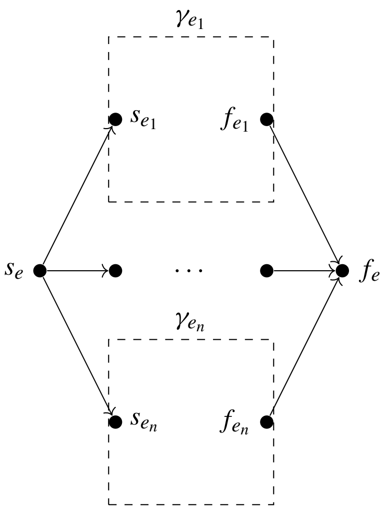
- If $e = a \in \Sigma' \cup \left\{\underline{\varepsilon}\right\}$, $\gamma_e = s_e \bullet \xrightarrow{\quad a \quad} \bullet f_e$;
- If $e = e' \cdot e''$, $\gamma_e =$



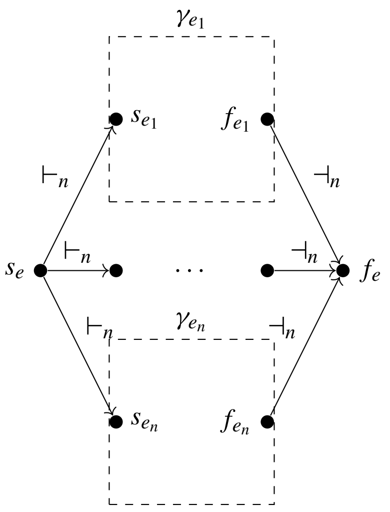
$\gamma_{e'}$
$\gamma_{e''}$
s_e
$\underline{\varepsilon}$
$s_{e'}$
$f_{e'}$
$\underline{\varepsilon}$
$s_{e''}$
$f_{e'}$
$\underline{\varepsilon}$
f_e ;

- If $e = \times\left(e'\right)$, $\gamma_e =$



– If $e = \vee(e_1, \ldots, e_n)$, $\gamma_e =$ 

.

– If $e = ||(e_1, \ldots, e_n)$, $\gamma_e =$ 

.

As an exemple, the IDL graph associated with the IDL expression describing the valid permutations of *Marcus cum amico caro ambulat* is:[6]

The parsing process of sentence *Caro cum amico Marcus ambulat* with the IDL graph presented in Fig. 9 is given in Fig. 10.

Given an IDL expression and its IDL graph, we also define the set of its cuts, that will serve as states in the parsing algorithm.

Definition 11 (*Cuts of an IDL expression*) Let $G = (N, \delta, \Sigma, F, P, S)$ be an IDL-PMCFG.

Let $(f, A_1, \ldots, A_q, A) \in P$. Let e be an IDL expression over $\Sigma' = \Sigma \cup \{X_{ij} \mid i \leq a(f), j \leq d_i(f)\}$, and γ_e its IDL graph.

The set of cuts of e, $C_e \subset \mathscr{P}(V(\gamma_e))$, where $V(\gamma_e)$ denotes the set of vertices of γ_e, is defined by induction as follows:

[6] Transitions labelled by $\underline{\varepsilon}$ are kept unmarked.

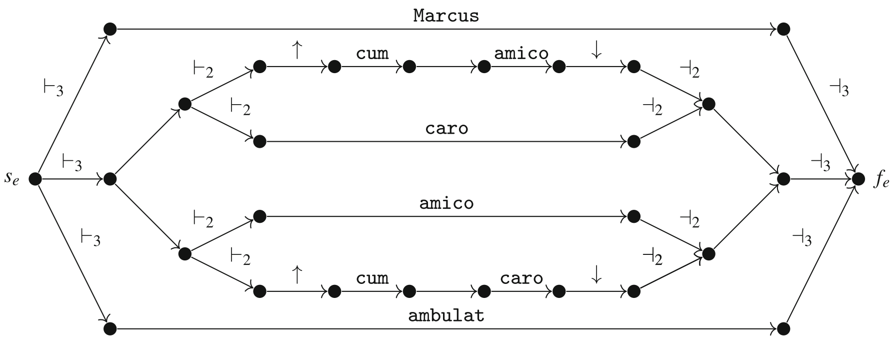


Fig. 9 IDL graph for *Marcus cum amico caro ambulat*

- If $e = a \in \Sigma' \cup \{\underline{\varepsilon}\}$, $C_e = \{\{s_e\}, \{f_e\}\}$;
- If $e = e' \cdot e''$, $C_e = \{\{s_e\}, \{f_e\}\} \cup C_{e'} \cup C_{e''}$;
- If $e = \times (e')$, $C_e = \{\{s_e\}, \{f_e\}\} \cup C_{e'}$;
- If $e = \vee (e_1, \ldots, e_n)$, $C_e = \{\{s_e\}, \{f_e\}\} \cup \bigcup_{k=1}^{n} C_{e_k}$;
- If $e = || (e_1, \ldots, e_n)$, $C_e = \{\{s_e\}, \{f_e\}\} \cup \prod_{k=1}^{n} C_{e_k}$, where $\prod$ denotes *n*ary Cartesian product.

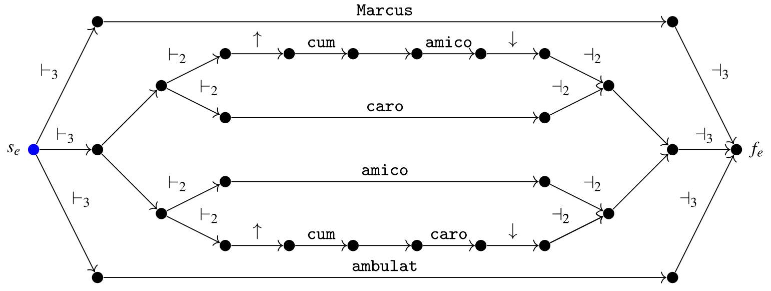
IDL graphs can be regarded as automata recognizing a given regular expression by reading it left-to-right, allowing for parallel reading of several interleaved substrings. The initial and final cuts are composed of a single node. *Split edges* marked by $\vdash_n$ cause several branches to be explored in parallel (thus increasing the cardinality of the current cut by $n - 1$ elements) while *merge edges* marked by $\dashv_n$ allow n nodes in the old cut to be replaced by one single node in the new cut. *Labelled edges* can be used to replace the node on the left-hand-side of the edge by the node on the right-hand-side of the edge in the current cut, provided that the terminal or nonterminal labelling the edge can be read at the current position. *Epsilon-labelled edges* (aka ε-transitions) can be taken under no specific assumption, provided that the left-hand-side node of the edge is in the current cut. They are especially used to encode disjunction nodes, which do *not* result in several branches being taken at the same time, but in only one of them to be chosen. The special *lock edges*, which were absent from Nederhof and Satta's original publication, will be discussed later.

An additional degree of complexity has to be dealt with in the context of IDL-PMCFG: we have to check that the substrings matched by the various nonterminals previously read are compatible with the constraints imposed on word order or interleaving. Therefore, throughout the execution of the algorithm, the current state of the parsing process within each IDL graph must be cautiously saved. Any field of any input category can match an arbitrary (and non necessarily contiguous) substring of the input. Moreover, given the ability of the formalism to encode nested lock constructions, an arbitrary number of such position sets must be remembered. This suggests the state space presented in Definition 13.

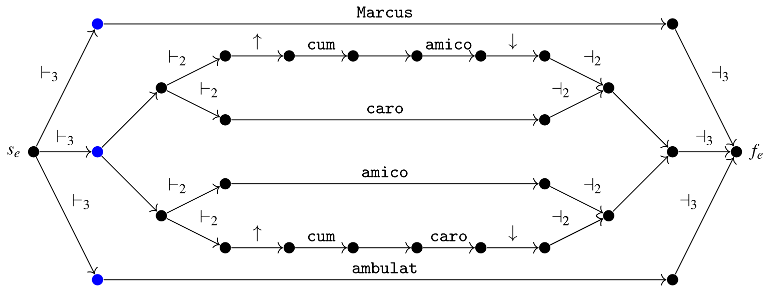
We first formally define a notion of stacks above an arbitrary set.

Current cut is marked in blue.

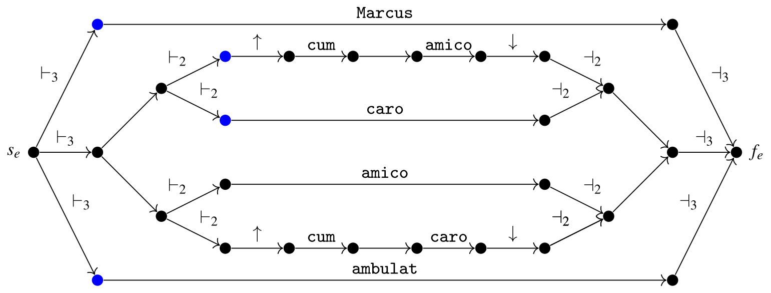
Step 1



Step 2 Read nothing, take *split* node.



Steps 3 and 4 Read nothing, take top ε-transition, then read nothing, take *split* node.



Step 5 Read *caro*.

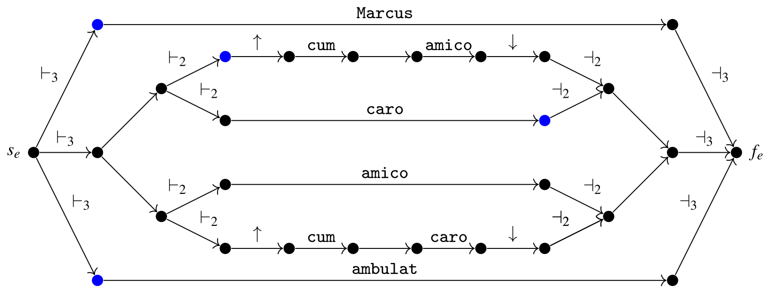
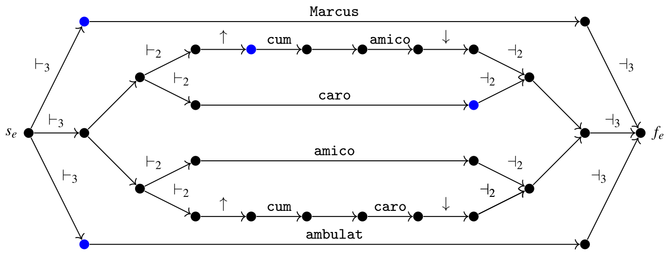
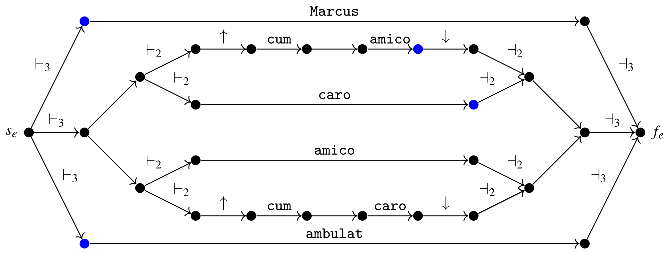


Fig. 10 Example of parsing *Caro cum amico Marcus ambulat* with the IDL graph from Fig. 9

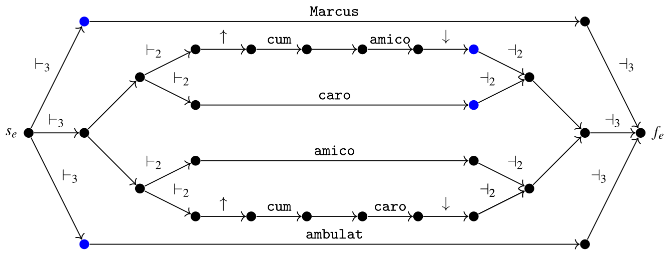
Step 6 Read nothing, take ↑-transition, start saving positions on current branch (initial value: ∅).



Steps 7, 8, 9 Read *cum*, update current position set on locked branch to $\{1\}$; read nothing, take ε-transition; read *amico*, update current position set on locked branch to $\{1,2\}$.



Step 10 Check that current position set on locked branch (i.e. $\{1,2\}$) is an interval, succeed and take ↓-transition.



Step 11 Check that positions matched by the active *cum* and *caro* branches are compatible (no word matched twice), succeed and take *merge* node.

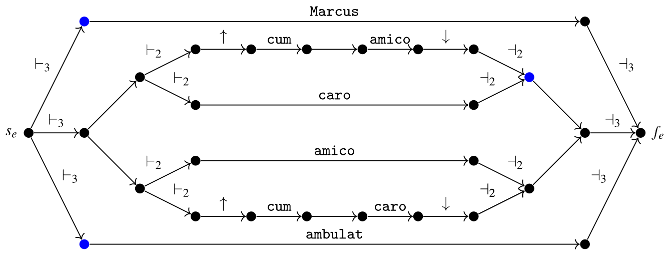
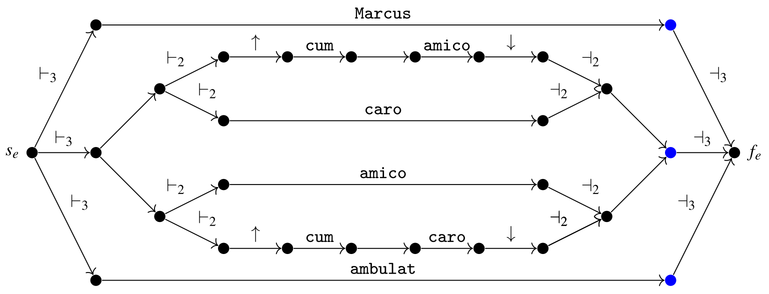


Fig. 10 continued

Steps 12, 13, 14, 15 Read nothing, take ε-transition; read *Marcus*; read *ambulat*.



Step 16 Check that positions matched by the 3 active branches are compatible, succeed and take *merge* node; terminate.

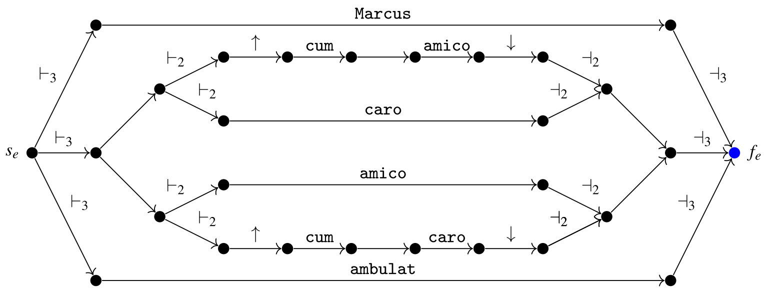


Fig. 10 continued

Definition 12 (*Stack over a set*) For any set S, $\mathsf{Stack}(S)$ is the set of stacks over S endowed with the two canonical primitives

$$\begin{aligned}\mathsf{pop} : \mathsf{Stack}(S) &\rightarrow (\mathsf{Stack}(S) \times S)\,,\\ h :: t &\mapsto (t, h)\\ \mathsf{push} : (\mathsf{Stack}(S) \times S) &\rightarrow \mathsf{Stack}(S)\\ (s, e) &\mapsto e :: s\end{aligned}$$

and an additional primitive defined only on non-empty stacks (see also Fig. 11).

$$\begin{aligned}\mathsf{applyHead} : (\mathsf{Stack}(S) \times (S \rightarrow S)) &\rightarrow \mathsf{Stack}(S)\\ (h :: t, f) &\mapsto f(h) :: t\end{aligned}$$

The state space of an IDL expression can now be defined.

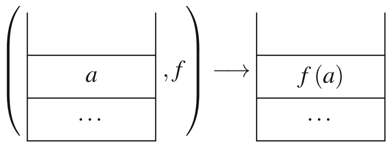


Fig. 11 Effect of primitive applyHead

Definition 13 (*State space of an IDL expression*) Let $G = (N, \delta, \Sigma, F, P, S)$ be an IDL-PMCFG.

Let $\left(f, A_1, \ldots, A_q, A\right) \in P$. Let e be an IDL expression over $\Sigma' = \Sigma \cup \left\{A_{ij} \mid i \leq a\left(f\right), j \leq d_i\left(f\right)\right\}$ and C_e its cuts.

The *general* state space of e is defined as

$$\mathscr{S}_e := \bigcup_{c \in C_e} \left\{(c, \vec{s}) \mid \vec{s} \in \left(\mathsf{Stack}\left(\mathscr{P}_f\left(\mathbb{N}\right)\right)\right)^c\right\}$$

and, for $t \in \Sigma^*$, the t-*specific* state space of e as

$$\mathscr{S}_{e,t} := \bigcup_{c \in C_e} \left\{(c, \vec{s}) \mid \vec{s} \in \left(\mathsf{Stack}\left(\mathscr{P}_f\left([\![1, |t|]\!]\right)\right)\right)^c\right\}$$

where $\mathscr{P}_f\left(\mathbb{N}\right)$ denotes the set of finite subsets of $\mathbb{N}$.

Informally, each state of an IDL parsing item is a pair $(c, \vec{\sigma})$ where c is a cut and σ is a map from the nodes of this cut to stacks of position sets. These stacks store the position of the terminals (words) that have already been read by the automaton when the current state is reached. Using a stack allows us to distinguish word positions that were matched in the current branch or at the current level of nested *lock*s, as opposed to words matched before the last split or outside of the current level of nested *lock*s. When a set of *split* transitions are taken, each of the new nodes added to the cut will store an independent copy of the previous stack, extended with an $\emptyset$ head. During the processing of the current branch, positions matched in the same branch will be added to the head of the stack, while non-head elements will store information from previous branches. When a set of *merge* transitions are taken, the heads of the various stacks will be first merged together (ensuring that no contradiction occurs) and then with the second element of all stacks (to take into account the closing of the current parallel processing and check, again, that no impossibility arises). With this technique, we can also give a simple semantics to the $\uparrow$ and $\downarrow$ edges: when an $\uparrow$ transition is taken, an $\emptyset$ is added to the current stack; when an $\downarrow$ transition is taken, we check whether the head element of the current stack is an interval and, if it is the case (and no incompatibility arises), we merge it with the second element.

The *incompatibilities* we evoked can be of two sorts: either the same positions have been read in two different branches, which can therefore not be merged; or what has been parsed does not respect the principle that an IDL graph, within the same branch, parses its input from left to right.

To formalize this, we introduce a partial order on sets of positions as well as some useful predicate:

Definition 14 The relation $\prec \subset \mathscr{P}_f\left(\mathbb{N}\right)^2$ is defined by

$$\forall\left(A, B\right) \in \mathscr{P}_f\left(\mathbb{N}\right)^2, A \prec B \Leftrightarrow \forall\left(a, b\right) \in A \times B, a < b.$$

Note that for any $A \in \mathscr{P}_f\left(\mathbb{N}\right)$ and $\emptyset \prec A$, $A \prec \emptyset$, and that moreover $\emptyset \prec \emptyset$.

Definition 15 The predicate $\mathsf{interval} \subset \mathscr{P}_f(\mathbb{N})$ is defined by

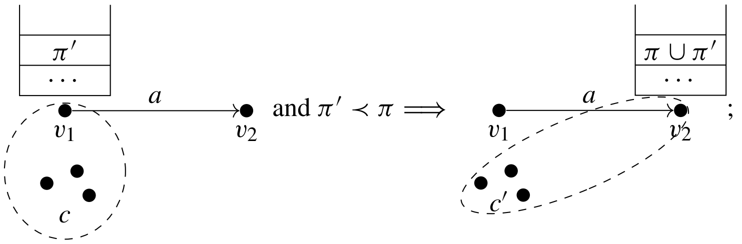
$$\forall A \in \mathscr{P}_f(\mathbb{N}), \mathsf{interval}(A) \Leftrightarrow \exists (a, b) \in \mathbb{N}, A = \{a, \dots, a + b\}.$$

We can finally define a transition relation between states:

Definition 16 (*Transition relation of an IDL expression*) Let $G = (N, \delta, \Sigma, F, P, S)$ be an IDL-PMCFG.

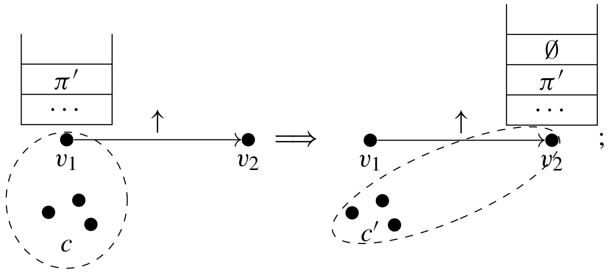
Let $(f, A_1, \dots, A_q, A) \in P$. Let e be an IDL expression over $\Sigma' = \Sigma \cup \{X_{ij} \mid i \le a(f), j \le d_i(f)\}$, $\gamma_e = (V_e, E_e)$ its IDL graph, C_e its cuts and $\mathscr{S}_e$ its states. The relation $\Delta_e \subset S_e \times \Sigma' \times \mathscr{P}_f(\mathbb{N}) \times S_e =: \Omega \times S_e$ is the smallest relation verifying the following axioms:

1. For all $(s = (c, \vec{s}), a, \pi) \in \Omega$, if there exists $(v_1, v_2, r) \in V_e^2 \times \mathscr{P}(V_e)$ such that $c = r \sqcup \{v_1\}$, $v_1 \xrightarrow{a} v_2 \in E_e$ and $\mathsf{pop}(\vec{s}(v_1))_1 \prec \pi$, then $\Delta_e(s, a, \pi, (r \cup \{v_2\}, \vec{s}'))$ where $\vec{s}' \in \mathsf{Stack}(\mathscr{P}_f(\mathbb{N}))^{r \cup \{v_2\}}$, $\vec{s}' |_r = \vec{s} |_r$ and $\vec{s}(v_2) = \mathsf{applyHead}(\vec{s}(v_1), \pi' \mapsto \pi \cup \pi')$; graphically:



this axiom encodes the fact that, when reading the (non)terminal a at position set $\pi \succ \pi'$ (meaning that π is located right of the previously read position set π'), we can update the current cut by replacing the node on the LHS of any edge labelled with a by the node on its RHS and appending the position set π to the positions stored on the top of the stack.

2. For all $(s = (c, \vec{s}), \underline{\varepsilon}, \emptyset) \in \Omega$, if there exists $(v_1, v_2, r) \in V_e^2 \times \mathscr{P}(V_e)$ such that $c = r \sqcup \{v_1\}$ and $v_1 \xrightarrow{\uparrow} v_2 \in E_e$, then $\Delta_e(s, a, \pi, (r \cup \{v_2\}, \vec{s}'))$ where $\vec{s}' \in \mathsf{Stack}(\mathscr{P}_f(\mathbb{N}))^{r \cup \{v_2\}}$, $\vec{s}' |_r = \vec{s} |_r$ and $\vec{s}(v_2) = \mathsf{push}(\vec{s}(v_1), \emptyset)$; graphically:

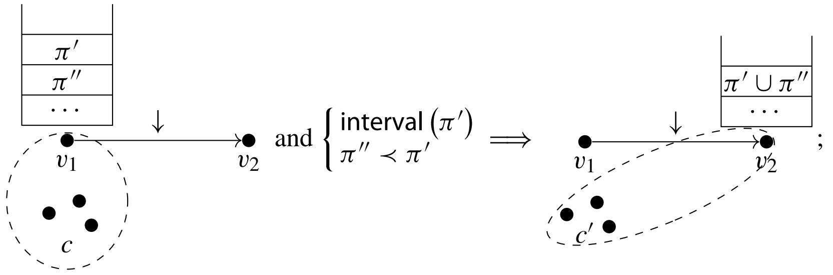


an $\uparrow$-edge can always be used to replace the node on the LHS of the edge by the node on its RHS, pushing an empty position set on the top of the corresponding stack —this is used to isolate the parsing of locked subexpressions, which are finally tested for connexity through a $\downarrow$-edge;

3. For all $\left(s = (c, \vec{s}), \underline{\varepsilon}, \emptyset\right) \in \Omega$, if there exists $(v_1, v_2, r) \in V_e^2 \times \mathscr{P}(V_e)$ such that $c = r \sqcup \{v_1\}, v_1 \xrightarrow{\downarrow} v_2 \in E_e$, $\mathsf{interval}\,(\mathsf{pop}\,(\vec{s}\,(v_1))_1)$ and $\mathsf{pop}\,(\mathsf{pop}\,(\vec{s}\,(v_1))_0)_1 \prec \mathsf{pop}\,(\vec{s}\,(v_1))_1$, then $\Delta_e\left(s, a, \pi, \left(r \cup \{v_2\}, \vec{s}'\right)\right)$ where $\vec{s}' \in \mathsf{Stack}\left(\mathscr{P}_f(\mathbb{N})\right)^{r \cup \{v_2\}}$, $\vec{s}' \mid_r = \vec{s} \mid_r$ and

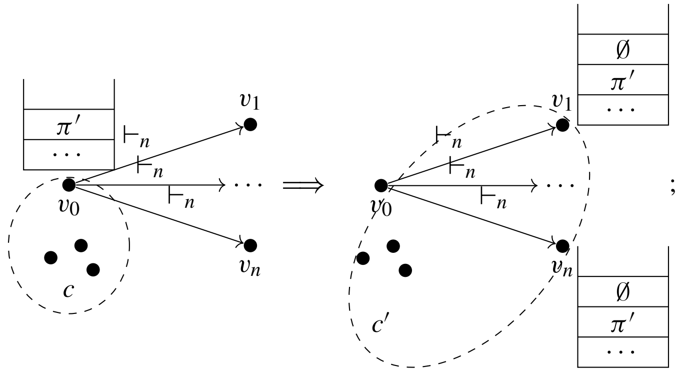
$$\vec{s}\,(v_2) = \mathsf{push}\left(\mathsf{pop}\,(\mathsf{pop}\,(\vec{s}\,(v_1))_0)_0\,, \mathsf{pop}\,(\mathsf{pop}\,(\vec{s}\,(v_1))_0)_1 \cup \mathsf{pop}\,(\vec{s}\,(v_1))_1\right);$$

graphically:



the $\downarrow$-edges are used at the end of locked subexpressions: the position set on the top of the stack, which stores the positions used in the current locked branch, is tested for connexity (with the $\mathsf{interval}$ primitive) and linear precedence (the newly closed locked branch at positions π must be located right of the previously read positions π''), which, if both tests succeed, leads to the node on the LHS of the edge to be replaced on its RHS and to both position sets to be merged;

4. For all $\left(s = (c, \vec{s}), \underline{\varepsilon}, \emptyset\right) \in \Omega$, if there exists $n \in \mathbb{N}$, $(v_0, v_1, \ldots, v_n, r) \in V_e^{n+1} \times \mathscr{P}(V_e)$ such that the v_i are distinct, $c = r \sqcup \{v_0\}$ and $\forall i \in \{1, \ldots, n\}, v_0 \xrightarrow{\vdash_n} v_i \in E_e$, then $\Delta_e\left(s, a, \pi, \left(r \cup \{v_1, \ldots, v_n\}, \vec{s}'\right)\right)$ where $\vec{s}' \in \mathsf{Stack}\left(\mathscr{P}_f(\mathbb{N})\right)^{r \cup \{v_1, \ldots, v_n\}}$, $\vec{s}' \mid_r = \vec{s} \mid_r$ and $\forall i \in \{1, \ldots, n\}, \vec{s}\,(v_2) = \mathsf{push}\,(\vec{s}\,(v_0)\,, \emptyset)$; graphically:



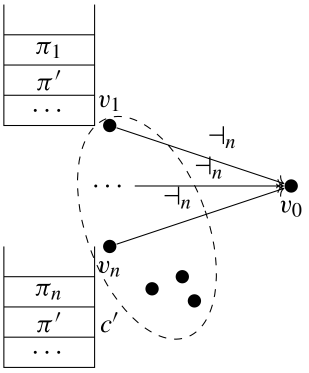
as soon as the current cut contains the LHS of a set of *split* (i.e. $\vdash_n$) edges, this axiom opens n parallel (interleaved) branches, replacing the LHS node by n RHS nodes, all of which come with the same stack as previously, except for an additional empty position set on top, which will later isolate the positions read in the various parallel branches;

5. For all $\left(s = (c, \vec{s}), \underline{\varepsilon}, \emptyset\right) \in \Omega$, if there exists $n \in \mathbb{N}$, $(v_0, v_1, \ldots, v_n, r) \in V_e^{n+1} \times \mathscr{P}(V_e)$ such that:
   - The v_i are distinct,
   - We have $c = r \sqcup \{v_1, \ldots, v_n\}$,
   - For all $i \in \{1, \ldots, n\}$, $v_i \xrightarrow{\dashv_n} v_0 \in E_e$,
   - For all $(i, j) \in \{1, \ldots, n\}^2$, $i \neq j \Rightarrow \mathsf{pop}(\vec{s}(v_i))_1 \cap \mathsf{pop}\left(\vec{s}\left(v_j\right)\right)_1 = \emptyset$,
   - We have $\mathsf{pop}(\mathsf{pop}(\vec{s}(v_1))_0)_1 \prec \bigsqcup_{i=1}^{n} \mathsf{pop}(\vec{s}(v_i))_1 =: m$;

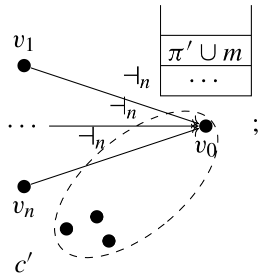
   then $\Delta_e\left(s, a, \pi, \left(r \cup \{v_0\}, \vec{s}'\right)\right)$ where $\vec{s}' \in \mathsf{Stack}\left(\mathscr{P}_f(\mathbb{N})\right)^{r \cup \{v_0\}}$, $\vec{s}' |_r = \vec{s} |_r$ and

$$\vec{s}(v_0) = \mathsf{applyHead}(\mathsf{pop}(\vec{s}(v_1))_0, \pi \mapsto \pi \cup m);$$

   graphically:



and $\begin{cases} \forall i \neq j, \pi_i \cap \pi_j = \emptyset \\ \pi' \prec \underbrace{\bigsqcup_{i=1}^{n} \pi_i}_{m} \end{cases}$ $\Longrightarrow$



;

   at the end of a series of parallel branches marked with *merge* (i.e. $\dashv_n$) edges (closing the parsing of an || node), this axiom checks that the position sets matched by the various parallel branches are compatible (disjunct) and that these positions, when merged, are compatible with previously matched positions (i.e. located right of them), and, in this case, it replaces the set of nodes on the LHS by the single RHS of merge edges.

Although these rules would essentially suffice to describe the parsing algorithm if all non-terminals appearing in a rule appeared exactly once, the fact that the same non-terminal may appear several times (copy) or not appear at all (erasement) requires us to keep trace of partial parsing contexts in which each argument may or may not have been already identified. We do this by introducing so-called *context tables*.

A context table is a partial function that associates to some arguments of the rule a partial mapping between some fields of these arguments and positions sets. It helps us remember which arguments have already been fixed and which ones can still be chosen freely. For each argument that has already been fixed, it retains which of its fields are available and what positions in the input string are matched by each available field.

Definition 17 (*Context table*) Let $G = (N, \delta, \Sigma, F, P, S)$ be an IDL-PMCFG.

Let $\left(f, A_1, \ldots, A_q, A\right) \in P$. Let e be an IDL expression over $\Sigma' = \Sigma \cup \left\{X_{ij} \mid i \leq a(f), j \leq d_i(f)\right\}$, $\gamma_e = (V_e, E_e)$ its IDL graph, C_e its cuts and $\mathscr{S}_e$ its states.

We call context table for e any $\Gamma \in [\![1, q]\!] \rightharpoonup \left(\mathbb{N} \rightharpoonup \mathscr{P}_f(\mathbb{N})\right)$ such that for all $i \in [\![1, q]\!]$, $\mathscr{D}(\Gamma(i)) \subset [\![1, \delta\left(A_q\right)]\!]$. The set of context tables for e is denoted by $\mathscr{G}_e$.

The set of context tables is equipped with three primitives defined as follows:

- For any input string s, $\mathsf{compat}_s \subset \mathscr{G}_e \times \mathbb{N}^2 \times \left(\mathbb{N} \rightharpoonup \mathscr{P}_f(\mathbb{N})\right) =: \Xi$ is such that

$$\forall (\Gamma, k, \ell, \vec{r}) \in \Xi,\ \mathsf{compat}_s(\Gamma, k, \ell, \vec{r}) \Leftrightarrow \begin{cases} \forall u \in \mathscr{D}(\vec{r}), \forall v \in \mathscr{D}(\Gamma), \forall w \in \mathscr{D}(\Gamma(v)), \vec{r}_u \cap \Gamma(v)_w = \emptyset & \text{if } k \notin \mathscr{D}(\Gamma) \\ \ell \in \mathscr{D}(\Gamma(k)) \cap \mathscr{D}(\vec{r}) \text{ and } s_{\Gamma(k)_\ell} = s_{\vec{r}_\ell} & \text{otherwise} \end{cases};$$

  in other terms, compat checks that the current context table Γ is compatible with mapping field ℓ of argument k to position set $\vec{r}_\ell =: \pi$, which is the case iff either (i) [*new nonterminal* (k, ℓ) *matched*] k is not in the context and π does not intersect any of the position sets stored in Γ or (ii) [*copy of already matched nonterminal*] k is already in the context, mapped to a partial function $\Gamma(k)$, which itself maps field index ℓ to a position set $\Gamma(k)_\ell =: \pi'$ such that substring s_π is the same as $s_{\pi'}$;
- The map $\mathsf{reserve} \in \Psi := \mathscr{G}_e \times \mathbb{N} \times \left(\mathbb{N} \rightharpoonup \mathscr{P}_f(\mathbb{N})\right) \to \mathscr{G}_e$ is defined as

$$\forall (\Gamma, k, \vec{r}) \in \Psi, \mathsf{reserve}(\Gamma, k, \vec{r}) = \begin{cases} \left[\bigcup_{u \in \mathscr{D}(\Gamma)} \{u \mapsto \Gamma(u)\}\right] \cup \{k \mapsto \vec{r}\} & \text{if } k \notin \mathscr{D}(\Gamma) \\ \Gamma & \text{otherwise} \end{cases};$$

  this map registers k in Γ, mapping it to $\vec{r}$, iff k is not yet in the context;
- For any input string s, $\mathsf{unify}_s \subset \mathscr{G}_e^3$ is such that

$$\forall \left(\Gamma, \Gamma', \Gamma''\right) \in \mathscr{G}_3, \mathsf{unify}_s\left(\Gamma, \Gamma', \Gamma''\right) \Leftrightarrow \forall k \in \mathscr{D}(\Gamma), \forall \ell \in \mathscr{D}(\Gamma(k)), \mathsf{compat}_s\left(\Gamma', k, \ell, \Gamma(k)\right) \wedge \left[\Gamma'' = \bigcup_{k \in \mathscr{D}(\Gamma')} \mathsf{reserve}\left(\Gamma, k, \Gamma'(k)\right)\right];$$

  primitive unify identifies triplets of contexts Γ, Γ' and Γ'' such that Γ'' can be obtained from Γ and Γ' by first (i) adding to Γ all matchings $k \mapsto \vec{r}$ from Γ' for which k is outside of the domain of Γ and then (ii) checking that for all k such that there exists $k \mapsto \vec{r} \in \Gamma$ and $k \mapsto \vec{r}' \in \Gamma'$, $\vec{r}$ and $\vec{r}'$ define the same fields and maps them to identical substrings.

The semantics of the three primitives are rather natural. First, compat indicates whether the assertion "the ℓth field of argument k can be identified in the input string at position set $\vec{r}_\ell$" is compatible with all prior decision stored in the current context. This is possible iff either the kth argument has never been matched before, or the ℓth field of the already detected kth argument in the current context is identical to the one matched by position set $\vec{r}_\ell$. Once a compatibility is detected, reserve is used to update the context table according to the newly matched item. Finally, unify allows us to compute the union of two contexts that do not interfere with each other.

4.2 Parsing COMPĀ Grammars

Our algorithm is inspired by Earley-style parsers designed to parse context-free GF or PMCFG grammars (Angelov et al. 2009; Ljunglöf 2012). The input is read from left to right and three different kinds of items are build bottom-up. The structure of the above context tables, that contain all essential information about the arguments of each rule and their position, makes it easier to recursively reconstruct all valid parse trees of a given input.

The three types of items we use are:

Definition 18 (*Parsing items*) Let $G = (N, \delta, \Sigma, F, P, S)$ be an IDL-PMCFG.

- *Active items* are items of the form $[\phi; e; s; \Gamma]$ where $\phi = \left(f, A_1, \ldots, A_q, A\right) \in P$, e is an IDL expression over $\Sigma' = \Sigma \cup \bigcup_{i=1}^{q} \left\{A_{ij} \mid j \in \{1, \ldots, d_i(f)\}\right\}$, $s \in \mathscr{S}_e$ and $\Gamma \in \mathscr{G}_e$;
- *Passive items* are items of the form $[\phi; A_i; r; \Gamma]_{\mathrm{P}}$ where $\phi = \left(f, A_1, \ldots, A_q, A\right) \in P$ and $\Gamma \in \mathscr{G}_e$, with e any IDL expression over $\Sigma' = \Sigma \cup \bigcup_{i=1}^{q} \left\{A_{ij} \mid j \in \{1, \ldots, d_i(f)\}\right\}$;
- *Completed items* are items of the form $\left[\phi; A; \vec{r}; \Gamma\right]_{\mathrm{C}}$ where $\phi = \left(f, A_1, \ldots, A_q, A\right) \in P$, $\vec{r} \in [\![1, q]\!] \rightharpoonup \mathscr{P}_f(\mathbb{N})$ and $\Gamma \in \mathscr{G}_e$, with e any IDL expression over $\Sigma' = \Sigma \cup \bigcup_{i=1}^{q} \left\{A_{ij} \mid j \in \{1, \ldots, d_i(f)\}\right\}$.

While active items store the current parsing status of a given IDL expression and passive items memorize successful parsing of a given IDL expression, completed items unify parsing results for different fields of the same category, checking that the various contexts are compatible with each other. Passive items are not absolutely essential; they are essentially syntactic sugar for active items where the current cut is reduced to the final node.

The deduction-style rules that make up the core of our algorithm are presented in Fig. 12. PREDICT, SCAN and COMBINE have their usual semantics from bottom-up parsing algorithms, and make extensive use of the context tables and transition relations defined in Sect. 4.1. STEP explores ε-transitions. SAVE produces a passive item from an active item that has reached it final state; this passive item is immediately converted into a completed item with only one activated field by SINGLETON. Finally, when a passive item can be used to extend the domain of a preexisting completed item, UNIFY performs this operation and returns a new completed item.

Predict

$$\frac{}{[\phi;e_i;\{s_{e_i}\mapsto\emptyset\};\emptyset]}\ \left(\left(e_1,\ldots,e_{\delta(A)}\right),A,A^1,\ldots,A^{a(n)}\right)\equiv\phi\in F$$

Scan

$$\frac{[\phi;e;s;\Gamma]\qquad \Delta_e\left(s,a,[\![j,j+1[\![,s'\right)}{[\phi;e;s';\Gamma]}\ a=t_j$$

Combine

$$\frac{[\phi;e;s;\Gamma]\qquad \left[\phi;A^k;\mathbf{r};\Gamma'\right]_{\mathrm{C}}\qquad \mathsf{compat}_s\left(\Gamma,k,\ell,\mathbf{r}\right)\qquad \Delta_e\left(s,(k,\ell),\mathbf{r}[\ell],s'\right)}{[\phi;e;s';\mathsf{reserve}\left(\Gamma,k,\mathbf{r}\right)]}$$

Step

$$\frac{[\phi;e;s;\Gamma]\qquad \Delta_e\left(s,\underline{\varepsilon},\emptyset,s'\right)}{[\phi;e;s';\Gamma]}$$

Save

$$\frac{[\phi;e_i;\{f_{e_i}\mapsto[r]\};\Gamma]}{[\phi;A_i;r;\Gamma]_{\mathrm{P}}}\ \left(\left(e_1,\ldots,e_{\delta(A)}\right),A,A^1,\ldots,A^{a(n)}\right)\equiv\phi\in F$$

Singleton

$$\frac{[\phi;A_i;r;\Gamma]_{\mathrm{P}}}{[\phi;A;\{i\mapsto r\};\Gamma]_{\mathrm{C}}}$$

Unify

$$\frac{[\phi;A;\mathbf{r};\Gamma]_{\mathrm{C}}\qquad [\phi;A_i;r';\Gamma']_{\mathrm{P}}\qquad i\notin\mathscr{D}(\mathbf{r})\qquad \mathsf{unify}_s\left(\Gamma,\Gamma',\Gamma''\right)}{[\phi;A;\mathbf{r}\cup\{i\mapsto r'\};\Gamma'']_{\mathrm{C}}}$$

Fig. 12 Parsing rules

Fig. 13 Item types and rules

$$\begin{tikzcd}
 & A\times C \arrow[rr, "\textsc{Combine}"] & & A \arrow[loop right, "\textsc{Scan,Step}"] \arrow[lll, dl, "\textsc{Save}"] \arrow[ll, dashed, bend left] & \\
P \arrow[dr, "\textsc{Singleton}"'] \arrow[drr, dashed] & & & & \emptyset \arrow[ul, "\textsc{Predict}"'] \\
 & C \arrow[uu, dashed] \arrow[r, dashed, bend left] & C\times P \arrow[l, "\textsc{Unify}"] & &
\end{tikzcd}$$

Practical implementation of the parsing algorithm requires that an (efficient) ordering be defined on the rules to apply. This ordering must guarantee correctness (i.e. that all possible syntax trees can be output) as well as an acceptable running time.

The graph from Fig. 13 displays the seven deduction rules from Fig. 12 as functions from and to the sets of active (A), passive (P) and completed items (C), as well as products thereof. Dashed arrows are added between two sets X and Y whenever there exists Z such that $Y = X \times Z$ (red) or $Y = Z \times X$ (green).

This graph can be viewed as a kind of "recursive control flow diagram" for our algorithm. Each edge labelled with a rule name corresponds to a recursive call to a corresponding function, that will try to apply the rule using the item output by the previous successful function call; each dashed edge corresponds to a fold operation through the item set matching the right-hand-side of the destination type (for red arrows) or the left-hand side of the destination type (for green arrows). The rule PREDICT is applied only at the first iteration. At each iteration, the parameters j and $a = t_j$ used by SCAN are updated, reading the input from left to right, and a sequence of recursive calls takes place, building new rules that are appended to the existing parsing environment.

In fact, due to the structure of our parsing system, one of the arrows above is redundant: the red arrow r from C to $C \times P$ can be removed without altering the correctness of the algorithm, as long as, when handling a passive item, the fold operation suggested by the green arrow g from P to $C \times P$ is executed before the recursive call encoded in the SINGLETON arrow. Let us consider the first iteration where the red arrow r is taken. This can only occur when a new completed item c has just been created; this completed item has been itself generated by either of the SINGLETON or UNIFY rules. If it has been generated by SINGLETON, say from a passive item p, a recursive fold through the set C has already taken place *via* the green arrow g. That fold has added to the environment all completed elements that can be computed from p and any other available completed item. Now, for any available passive item p', a completed item c' has been derived from p' at some point of time in the past. The fold operation triggered by g has already, if possible, derived a new complete item from p and c' that would contain exactly the same information as the item to be created from c and p'. If c has been created by UNIFY, a fold has already been triggered through g (the only possible path to reach $C \times P$ has been through g, because of our hypothesis that we have not taken r before) and the same reasoning applies by considering the items c' and p used to produce c.

The resulting pseudocode is presented in Algorithm 2.

4.3 Complexity

The goal of the final part of this paper is to provide an upper bound of the complexity of our parsing algorithm under some practically reasonable assumptions, that will be obtained as Theorem 4. The detailed proof of this theorem can be found in Appendix A.1.

Before addressing the actual complexity problem, three remarks must be made.

First, $\vee$ nodes are not absolutely needed in the IDL-PMCFG formalism. By creating some new rules and introducing intermediate categories, it is easy to transform any grammar into an equivalent one without any $\vee$ node. In the following discussion, we will often exclude the case of $\vee$ nodes, and give upper bounds only for the case where those $\vee$ nodes are not used. Practical experience showed that disjunction, being redundant with the creation of two separate rules, are a useful, but less frequently used

Algorithm 2 COMPĀ's parsing algorithm

```
1: function Parse(t)
2:     $E \leftarrow \emptyset$
3:     PredictAll $(E)$
4:     for $j \in [\![0, |t|[\![$ do
5:         for $I = [\phi; e; s; \Gamma] \in E$ do
6:             TryScan $(I, a_j, j, E)$
7: function PredictAll(E)
8:     for $\left((e_1, \ldots, e_{\delta(A)}), A, A^1, \ldots, A^{a(n)}\right) \in F$ do
9:         for $i \in [\![1, \delta(A)]\!]$ do
10:            $I \leftarrow \left[\phi; e_i; \left\{s_{e_i} \mapsto [\emptyset]\right\}; \text{INIT}\left(e_i\right)\right]$
11:            $E \leftarrow E \cup \{I\}$                                    ▷ Apply rule Predict
12:            TryRec $(I, E)$
13: function TryScan($[\phi; e; s; \Gamma], a, j, E$)
14:    for $s' \in \mathscr{S}_e$ s.t. $\Delta_e\left(s, a, [\![j, j+1[\![, s'\right)$ do
15:        $I \leftarrow \left[\phi; e; s'; \Gamma\right]$
16:        $E \leftarrow E \cup \{I\}$                                        ▷ Apply rule Scan
17:        TryRec $(I, E)$
18: function TrySave($[\phi; e; s; \Gamma], E$)
19:    if $s = \left\{f_{e'} \mapsto [r]\right\}$ then
20:        for $\phi = \left((e_1, \ldots, e_{\delta(A)}), A, A^1, \ldots, A^{a(n)}\right) \in F, i \in [\![1, \delta(A)]\!], e_i = e'$ do
21:            $I \leftarrow \left[\phi; A_i; r; \Gamma\right]_{\mathrm{P}}$
22:            $E \leftarrow E \cup \{I\}$                                    ▷ Apply rule Save
23:            $I' \leftarrow [\phi; A; \{i \mapsto r\}; \Gamma]_{\mathrm{C}}$
24:            $E \leftarrow E \cup \left\{I'\right\}$                          ▷ Apply rule Singleton
25:            TryUnify $(I, E)$
26: function TryStep($[\phi; e; s; \Gamma], E$)
27:    for $s' \in \mathscr{S}_e$ s.t. $\Delta_e\left(s, \underline{\varepsilon}, \emptyset, s'\right)$ do
28:        $I \leftarrow \left[\phi; e; s'; \Gamma\right]$
29:        $E \leftarrow E \cup \{I\}$                                        ▷ Apply rule Step
30:        TryRec $(I, E)$
31: function TryCombineLHS($[\phi; e; s; \Gamma], E$)
32:    for $\left[\phi'; A^k; \vec{r}\right]_{\mathrm{C}} \in E$ do
33:        for $\ell \in \mathscr{D}(\vec{r}), s' \in \mathscr{S}_e$ s.t. $\Delta_e\left(s, (k, \ell), \vec{r}[\ell], s'\right)$ do
34:            if $\mathsf{compat}_t(\Gamma, k, \ell, \vec{r})$ then
35:                $I \leftarrow \left[\phi; e; s'; \mathsf{reserve}(\Gamma, k, \vec{r})\right]$
36:                $E \leftarrow E \cup \{I\}$                                ▷ Apply rule Combine
37:                TryRec $(I, E)$
38: function TryCombineRHS($\left[\phi'; A^k; \vec{r}; \Gamma'\right]_{\mathrm{C}}, E$)
39:    for $[\phi; e; s; \Gamma] \in E$ do
40:        for $\ell \in \mathscr{D}(\vec{r}), s' \in \mathscr{S}_e$ s.t. $\Delta_e\left(s, (k, \ell), \vec{r}[\ell], s'\right)$ do
41:            if $\mathsf{compat}_t(\Gamma, k, \ell, \vec{r})$ then
42:                $I \leftarrow \left[\phi; e; s'; \mathsf{reserve}(\Gamma, k, \vec{r})\right]$
43:                $E \leftarrow E \cup \{I\}$                                ▷ Apply rule Combine
44:                TryRec $(I, E)$
45: function TryUnify($\left[\phi; A^k_\ell; r; \Gamma\right]_{\mathrm{P}}, E$)
46:    for $\left[\phi; A^k; \vec{r}'; \Gamma'\right]_{\mathrm{C}} \in E$ do
47:        if $\ell \notin \mathscr{D}\left(\vec{r}'\right)$ and $\exists \Gamma'' \in \mathscr{G}_e, \mathsf{unify}_t\left(\Gamma, \Gamma', \Gamma''\right)$ then
48:            $I \leftarrow \left[\phi; A^k; \vec{r}' \cup \{\ell \mapsto r\}; \Gamma''\right]_{\mathrm{C}}$
49:            $E \leftarrow E \cup \{I\}$                                    ▷ Apply rule Unify
50:            TryCombineRHS $(I, E)$
51: function TryRec($I, E$)
52:    TrySave $(I, E)$
53:    TryStep $(I, E)$
54:    TryCombineLHS $(I, E)$
```

feature of the formalism. Nevertheless, we shall give some insights in Appendix A.1.3 about how to take into account disjunction in the final estimate.

Second, we introduce a notion of *G-density of a language*:

Definition 19 (*G-density of a language*) Let $m \in \mathbb{N}$. Let $G = (N, \delta, \Sigma, F, P, S)$ be an m-parallel IDL-MCF grammar. Let $T \subset \Sigma^*$ such that $\varepsilon \notin T$. The G-density of T is defined as

$$\rho_{T,G} = \sup_{A \in N} \sup_{i \in [\![1, \delta(A)]\!]} \sup_{t \in T} \frac{\left|\left\{p \in \mathscr{P}([\![1, |t|]\!]) \setminus \{\emptyset\} \mid A \hat{\rightarrow} \left\langle \alpha_1, \ldots, \alpha_{i-1}, t_p, \alpha_{i+1}, \ldots, \alpha_{\delta(A)} \right\rangle \right\}\right|}{|t|}$$
$$\in \mathbb{R}_+ \cup \{+\infty\}$$

where t_p denotes the substring of t composed of the tokens at positions p in t (see notations).

The G-density of a language serves as a proxy for the amount of ambiguity that this language contains from the point of view of grammar G. It answers the question: 'How many different substrings of any string in the language can be matched by the same field of the same category?'. This 'how many' is quantified as a quotient of the number of different matches over the length of any input string. Introducing G-density will allow us to discuss the worst-case complexity of parsing on *reasonable* sets of inputs, i.e. those for which ρ_T is finite, or, equivalently, for which the number of matched substrings grows at most linearly in the size of the string.

Consider the case where we want to describe adjective-noun attachment in a natural language where adjectives can be placed arbitrarily before or after the noun they modify. We are given an (arbitrary large) lexicon with a number of terminal rules producing adjectives (of category A) and nouns (of category N). These terminals are stored in an alphabet we denote by Σ. The part of the grammar building noun phrases (of category S) in our toy IDL-CF grammar looks like this:

$$N \rightarrow S : n \mapsto n$$
$$S \rightarrow A \rightarrow S : np, a \mapsto \,||\, (np, a) \,.$$

Now, how ambiguous can noun phrases be? If we take $T = \Omega := \mathscr{P}(\Sigma^*)$, then considering a string with a noun and n arbitrary adjectives in any order results in all substrings containing the noun to be valid noun phrases; in this case, $\rho_{T,G} \geq \frac{2^n}{n+1}$ for all $n \in \mathbb{N}^+$, and therefore $\rho_{T,G} = +\infty$. But if we now consider the (more practical) case where the number of adjectives to be attached to the same noun is less than some reasonable constant M, and call U the corresponding sublanguage of Ω, then we get no more than $2^{\min(k-1,M)}$ different matching substrings for any input of length $k \geq 1$; as a consequence, $\rho_{U,G} = \sup_{k \in \mathbb{N}^+} \frac{2^{\min(k-1,M)}}{k} = \frac{2^M}{M+1} < +\infty$.

Third, we need to keep in mind the following fact, that is an immediate consequence of Theorem 1:

Theorem 3 *IDL-PMCFG parsing is NP-complete. Therefore, unless* P = NP, *general IDL-PMCFG parsing is not polynomial in the size of the input string.*

Proof Theorem 1 provides a reduction from the NP-complete problem 3SAT to parsing IDL-PMCF the grammar G from Lemma 3 □

4.3.1 Measuring IDL Graphs

We now introduce two measures of the complexity of IDL graphs, encoded in two primitives height and width. While height was already defined in Nederhof and Satta's paper (though it was there called width, and defined in a slightly different manner), width plays a new and complementary role that we shall emphasize later. The informal interpretation of these metrics is simple: height measures the maximal number of branches that can be traversed in parallel, while width quantifies the maximal number of edges labelled with a terminal or $\underline{\varepsilon}$ on any left-to-right path from the start to the end node.

Definition 20 (*Height and width of an IDL expression graph*) Let Σ be a set of symbols that does not contain $\underline{\varepsilon}$ and $\diamond$ and $\mathfrak{E}$ the set of IDL expressions over Σ. The height and width of an IDL expression $e \in \mathfrak{E}$ are defined inductively as follows:

$$\begin{aligned}
\mathsf{height}\,(a) &= 1 && \forall a \in \Sigma \cup \left\{\underline{\varepsilon}\right\}\\
\mathsf{height}\left(e' \cdot e''\right) &= \max\left(\mathsf{height}\left(e'\right), \mathsf{height}\left(e''\right)\right)\\
\mathsf{height}\,(\times\,(e)) &= \mathsf{height}\,(e)\\
\mathsf{height}\,(\vee\,(e_1, \ldots, e_n)) &= \sum_{i=1}^{n} \mathsf{height}\,(e_i)\\
\mathsf{height}\,(||\,(e_1, \ldots, e_n)) &= \sum_{i=1}^{n} \mathsf{height}\,(e_i)\\
\mathsf{height}\left(||\left(e'\right)\right) &= \mathsf{height}\left(e'\right);\\
\mathsf{width}\,(a) &= 1 && \forall a \in \Sigma \cup \left\{\underline{\varepsilon}\right\}\\
\mathsf{width}\left(e' \cdot e''\right) &= \mathsf{width}\left(e'\right) + \mathsf{width}\left(e''\right)\\
\mathsf{width}\,(\times\,(e)) &= \mathsf{width}\,(e)\\
\mathsf{width}\,(\vee\,(e_1, \ldots, e_n)) &= \max_{i=0}^{n} \mathsf{width}\,(e_i)\\
\mathsf{width}\,(||\,(e_1, \ldots, e_n)) &= \max_{i=0}^{n} \mathsf{width}\,(e_i)\,.
\end{aligned}$$

In the graph γ of Fig. 9, we have $\mathsf{width}\,(\gamma) = 3$ (a path from left to right in the graph contains at most two edges labelled by terminals) and $\mathsf{height}\,(\gamma) = 6$ (there are at most six nodes in a cut, or equivalently six branches traversed in parallel).

4.3.2 Final Complexity Estimate

Based on the previous definitions of height and width, we can prove

Theorem 4 *Let $m \in \mathbb{N}$. Let $G = (N, \delta, \Sigma, F, P, S)$ be an m-parallel IDL-MCF grammar. Let E be the set of IDL expressions used in G. Assume that for all $e \in E$, e does not contain any $\vee$ node. Let $T \subset \Sigma^*$ and put $\rho := \rho_T$. Let $w := \max_{e \in E} \mathsf{width}(e)$, $h := \max_{e \in E} \mathsf{height}(e)$, $R = |P|$, $\alpha = \max_{f \in F} a(f)$, $M = \max_{e \in E} |e|$. Finally, let $t \in T$ and $n := |t|$. Assume that $w \geq 6$ and $n \geq w$. An upper bound of the complexity of the parsing algorithm described in Algorithm 2 is given by*

$$\mathscr{O}\left(\alpha m^2 R h \rho^m \left(\left(\frac{\rho w^2}{h^2}\right)^w \frac{1}{(w-1)!}\right)^h h^n n^{2hw+m+2}\right).$$

Proof See A.1. □

5 Conclusion

In this paper, we have presented and studied IDL-PMCFG, a new grammatical formalism that extends PMCFG with Nederhof and Satta's IDL expression. This formalism, along with its GF-like experimental front-end COMPĀ, was designed as a tool to formally encode the syntax of free word order languages. COMPĀ, its IDL-PMCFG backbone and the associated parsing algorithm have been implemented in an experimental setup, focussing on the parsing of Classical Latin. The corresponding code can be found in our GitHub repository. To our knowledge, this formalism is the only one to this day to allow for a straightforward, wide-coverage syntactic description of Classical Latin and similar languages for rule-based parsing purposes. The fact that IDL-PMCFG extends PMCFG with only two new operators should make extending existing tools for support of hyperbatic constructions comparatively smoother than if an ad hoc approach to Latin syntax had been chosen.

In order to be able to easily encode the kind of extensive free word order encountered in the case of Classical Latin, an operator allowing for grammatical constituents to be swapped and intertwined, the $||$ operator, is required; no less required for conciseness is the ability, in particular instances, to impose fixed constituent order (through the $\cdot$ operator) or non-interleaving, or locking, of constituents (through the $\times$ operator). Since these operators are virtually anavoidable when it comes to providing a linguistically intuitive description of the actual syntactic constraints in the language, it is reasonable to think of IDL-PMCFG as the "smallest extention of PMCFG with built-in support for free word order as observed in Classical Latin". Note that this does not mean that IDL-PMCFG would be the smallest extension of CFG with this same property, since copying is not required to encode hyperbatic constructions. Besides the design and analysis of the formalism itself, one of the main contributions of this paper is the classification result of Theorem 1. This theorem has two main consequences.

The first one is mainly theoretical: whenever a CFG-derived grammatical formalism is coupled with IDL expressions and includes a record system that does not restrict copying, parsing in this formalism must, in the worst case, be non-polynomial in the size of the input. As an immediate corollary, such formalisms cannot be mildly context-sensitive. In fact, even if we disallowed copying, there is no way a formalism able to generate Latin hyperbatic structures in a linguistically meaningful way could be mildly context sensitive: Becker et al. (1992) showed that scrambling as it occurs in German—a kind of free word order that is strictly less general than Latin hyperbata—is not mildly context-sensitive.

The second, more practical consequence is that the corresponding parsing algorithms will not be polynomial, which does of course *not* mean that parsing will be intractable altogether, since pratical linguistic settings rarely present the level of ambiguity that leads to theoretical worst cases. Our first experiments would rather suggest the opposite. Studying the complexity of the IDL-PMCFG parsing algorithm in the particular case of IDL-MCF grammars without copy would be an interesting path for further research.

While a majority of works in formal NLP draw most of their examples from fixed word order languages (most notably English, but also to certain extent German, French, or Chinese) in which hyperbata are almost always ungrammatical, it might be tempting to think that mild context sensitivity, and in particular polynomial-time parsing, is sufficiently expressive to account for syntactic phenomena in a vast majority of instances and for *almost all natural languages*. This view indeed seems to widely accepted,[7] and it has proved practical in many cases, often preventing unnecessary computational explosion.

Becker et al. (1992) showed that its theoretical accurateness should be questioned. Indeed, in the light of this study, the alleged *minimality* of mildly context-sensitive languages, while not contradicted by the grammar of English and similar languages, appears to have somewhat underestimated the complexity of (very) free word order: Classical Latin and Greek, Sanskrit etc. present hyperbatic constructions that are considerably more complex that German scrambling. These may well be specific languages, and, in one sense, they are: Classical Latin and Greek, as well as Sanskrit, belong to a rather extremal subset in the (somewhat imprecise) galaxy of so-called free word order languages. In these languages, audacious interleaves and permutations have become part of a canon of refined rhetorical and prosodic effects, thus enhancing even more the natural syntactic flexibility of a morphologically rich linguistic system. Many other idioms, such as English, are not concerned by this kind of phenomena, and it would be equally unsatisfying to impose free word order formalisms upon them. What is at stake is not so much the pertinence of mild context sentivity for the vast majority of formal NLP applications, but rather its *universality* throughout natural languages.

Two import questions remain open. On the formal side, the position of IDL-CFGs and IDL-MCFGs (without copy) in the hierarchy are still unclear, and so is the complexity of their respective parsing algorithms. On the linguistic side, the level of expressivity needed to account for hyperbaton and locking of clauses is not precisely known. Answering these two questions would provide a more complete insight into the level of syntactic complexity of free word order in natural language, while paving the way for the development of more efficient description and parsing systems.

[7] Following Joshi, Kallmeyer (2010) writes "there is still good reason to assume that natural languages are mildly context-sensitive" (p. 25).

Acknowledgements The research presented in this paper was conducted primarily during a research stay at the University of Gothenburg. I thank Prof. Aarne Ranta and Dr. Peter Ljunglöf (both University of Gothenburg and Chalmers University of Technology) for their time, advice and helpful discussions, and Dr. Prasanth Kolachina and Herbert Lange (both University of Gothenburg) for their support and suggestions. I am especially grateful to Dr. Peter Ljunglöf for his extensive remarks on the draft version of this work. Last but not least, my best thanks go to the two anonymous reviewers whose remarks greatly contributed to improving the quality of this paper.

Funding Open access funding provided by Swiss Federal Institute of Technology Zurich.

A Appendix

A.1 Complexity Estimate

In the next subsections, we estimate the complexity of Algorithm 2 building on the notions of height and width introduced in Sect. 4.3.1; we shall

A.1.1 Define *relaxation* of IDL expressions and prove that every IDL expression can be relaxed in such a way that height and width are preserved while the number of so-called *stable states* can only increase;
A.1.2 Give an upper bound for parsing with relaxed IDL expressions;
A.1.3 Finally, derive an upper bound of the complexity of the algorithm under some practical assumptions.

A few combinatorial lemmas that will be used in A.1.2 can be found in A.2.

A.1.1 Relaxed IDL Expressions and Stable States

We first introduce a primitive that sorts a list of IDL expressions not containing any $||$, $\vee$ or $\times$ node by order of descreasing width.

Definition 21 (sort *primitive*) Let Σ be a set of symbols that does not contain $\underline{\varepsilon}$ and $\diamond$. Let $\mathfrak{F}$ be the set of IDL expressions over Σ that do not contain any $||$, $\vee$ or $\times$ node.

Let $n \in \mathbb{N}^+$ and $(e_1, \ldots, e_n) \in \mathfrak{F}^n$. For $i \in [\![1, n]\!]$, it is clear that expression e_i is of the form $a_{i1} \cdot \cdots \cdot a_{ik_i}$ for some $k_i = \mathsf{width}\,(e_i) \geq 1$, where $a_i \in \Sigma \cup \left\{\underline{\varepsilon}\right\}$ for all $i \in [\![1, k_i]\!]$. Therefore, we can easily choose a permutation $\sigma_{(e_1,\ldots,e_n)} \in \mathfrak{S}_k$ such that

$$\forall\,(i, j) \in [\![1, n]\!]^2,\; i < j \Rightarrow k_{\sigma_{(e_1,\ldots,e_n)}(i)} \geq k_{\sigma_{(e_1,\ldots,e_n)}(j)}.$$

Let $m \in \mathbb{N}^+$. Given a choice of $\sigma_{(e_1,\dots,e_m)}$ for all $(e_1, \dots, e_m) \in \mathfrak{F}^m$, we define sort_m as

$$\mathsf{sort}_m : \mathfrak{F}^m \to \mathfrak{F}^m, (e_1, \dots, e_m) \mapsto \left(e_{\sigma_{e_1,\dots,e_m}(1)}, \dots, e_{\sigma_{e_1,\dots,e_m}(m)}\right).$$

Finally, we define sort as the only function from $\sqcup_{k\in\mathbb{N}^+}\mathfrak{F}^k$ into itself such that $\mathsf{sort}\,|_{\mathfrak{F}^k}= \mathsf{sort}_k$ for all $k \in \mathbb{N}^+$.

Now, it is time for us to introduce *relaxed IDL expressions*.

Definition 22 (*Relaxed IDL expression*) Let Σ be a set of symbols that does not contain $\underline{\varepsilon}$ and $\diamond$ and $\mathfrak{E}_*$ the set of IDL expressions over Σ that do not contain any $\vee$ node. The relaxation $\tilde{e}$ of an IDL expression $e \in \mathfrak{E}_*$ is defined inductively as follows:

$$\begin{aligned}
\widetilde{a} &= \|\,(a) && \forall a \in \Sigma \cup \{\mathscr{E}\}\\
\widetilde{e_1 \cdot e_2} &= \|\left(\mathsf{sort}\left(e'_{11}\cdot e'_{21}, \dots, e'_{1k_1}\cdot e'_{2k_1}, e'_{2(k_1+1)}, \dots, e'_{2k_2}\right)\right) && \text{where } \forall i \in \{1,2\},\\
\widetilde{e_i} &= \|\left(e'_{i1}, \dots, e'_{ik_i}\right), k_1 \le k_2\\
\widetilde{e_1 \cdot e_2} &= \|\left(\mathsf{sort}\left(e'_{11}\cdot e'_{21}, \dots, e'_{1k_2}\cdot e'_{2k_2}, e'_{2(k_2+1)}, \dots, e'_{2k_1}\right)\right) && \text{where } \forall i \in \{1,2\},\\
\widetilde{e_i} &= \|\left(e'_{i1}, \dots, e'_{ik_i}\right), k_1 \ge k_2\\
\widetilde{\times\left(e'\right)} &= \widetilde{e'}\\
\|\,\widetilde{(e_1, \dots, e_n)} &= \|\left(\mathsf{sort}\left(e'_{11}, \dots, e'_{nk_n}\right)\right) && \text{where } \forall i \in [\![1,n]\!],\\
\widetilde{e_i} &= \|\left(e'_{i1}, \dots, e'_{ik_i}\right).
\end{aligned}$$

Relaxed IDL expressions over Σ are of the form

$$e = \|\left(a_{11}\cdot\dots\cdot a_{1k_1}, \dots, a_{n1}\cdot\dots\cdot a_{nk_n}\right)$$

where $\forall i \in [\![1,n]\!] \forall j \in [\![1,k_i]\!]$, $a_{ij} \in \Sigma \cup \{\mathscr{E}\}$ and $k_1 \ge \dots \ge k_n$; in particular, $\mathsf{height}\,(e) = n$ and $\mathsf{width}\,(e) = k_1$.

A simple simultaneous induction is sufficient to prove both this fact and the well-definedness of relaxation.

Relaxation increases the number of substrings of the input that can be matched by an expression:

Lemma 4 *Let Σ be a set of symbols that does not contain $\underline{\varepsilon}$ and $\diamond$ and $\mathfrak{E}_*$ the set of IDL expressions over Σ that do not contain any $\vee$ node. Let $t \in \Sigma^*$. For all $p \in \mathscr{P}([\![1,|t|]\!])$, if t_p is recognized by e, then t_p is also recognized by $\widetilde{e}$.*

Proof By immediate induction on $e \in \mathfrak{E}_*$. □

We can now check that relaxation keeps height and width invariant:

Proposition 9 *Let Σ be a set of symbols that does not contain $\underline{\varepsilon}$ and $\diamond$ and e an IDL expression over Σ that does not contain $\vee$. We have $\mathsf{width}\,(\widetilde{e}) = \mathsf{width}\,(e)$ and $\mathsf{height}\,(\widetilde{e}) = \mathsf{height}\,(e)$.*

Proof We prove this by induction on $e \in \mathfrak{E}_*$:

- If $e = a \in \Sigma \cup \{\underline{\varepsilon}\}$, we check that $\mathsf{width}(e) = \mathsf{height}(e) = 1$, $\mathsf{width}(\widetilde{e}) = \mathsf{width}(||(a)) = \mathsf{width}(a) = 1$ and $\mathsf{height}(\widetilde{e}) = \mathsf{height}(||(a)) = \mathsf{height}(a) = 1$.
- If $e = e_1 \cdot e_2$, then let $(k_1, k_2) \in (\mathbb{N}^+)^2$, $\left(e'_{11}, \ldots, e'_{1k_1}, e'_{21}, \ldots, e'_{2k_2}\right) \in \mathfrak{E}_*^{k_1+k_2}$ such that $\widetilde{e_i} = ||\left(e'_{i1}, \ldots, e'_{ik_i}\right)$ for $i \in \{1, 2\}$. Without loss of generality, assume $k_2 \geq k_1$.
  By definition of relaxation, we have $\widetilde{e} = ||\left(\mathsf{sort}\left(e'_{11} \cdot e'_{21}, \ldots, e'_{1k_1} \cdot e'_{2k_1}, e'_{2(k_1+1)}, \ldots, e'_{2k_2}\right)\right)$.
  For $i \in \{1, 2\}$, we know that:

  1. By Definition 22, $\mathsf{width}(\widetilde{e_i}) = \mathsf{width}\left(e'_{i1}\right)$ and $\mathsf{height}(\widetilde{e_i}) = k_i$;
  2. By induction hypothesis, $\mathsf{width}(\widetilde{e_i}) = \mathsf{width}(e_i)$ and $\mathsf{height}(\widetilde{e_i}) = \mathsf{height}(e_i)$.

  There follows:

$$\begin{aligned}
\mathsf{width}(\widetilde{e}) &= \max\left(\mathsf{width}\left(e'_{11} \cdot e'_{21}\right), \ldots, \mathsf{width}\left(e'_{1k_1} \cdot e'_{2k_1}\right),\right.\\
&\quad \left.\mathsf{width}\left(e'_{2(k_1+1)}\right), \ldots, \mathsf{width}\left(e'_{2k_2}\right)\right)\\
&= \max\left(\mathsf{width}\left(e'_{11}\right) + \mathsf{width}\left(e'_{21}\right), \ldots, \mathsf{width}\left(e'_{1k_1}\right)\right.\\
&\quad \left.+\mathsf{width}\left(e'_{2k_1}\right), \mathsf{width}\left(e'_{2(k_1+1)}\right), \ldots, \mathsf{width}\left(e'_{2k_2}\right)\right)\\
&= \mathsf{width}\left(e'_{11}\right) + \mathsf{width}\left(e'_{21}\right)\\
&= \mathsf{width}(\widetilde{e_1}) + \mathsf{width}(\widetilde{e_2})\\
&= \mathsf{width}(e);\\
\mathsf{height}(\widetilde{e}) &= k_2\\
&= \max(k_1, k_2)\\
&= \max(\mathsf{height}(\widetilde{e_1}), \mathsf{height}(\widetilde{e_2}))\\
&= \max(\mathsf{height}(e_1), \mathsf{height}(e_2))\\
&= \mathsf{height}(e).
\end{aligned}$$

- If $e = \times(e')$, then by induction hypothesis $\mathsf{width}(\widetilde{e}) = \mathsf{width}(\widetilde{e'}) = \mathsf{width}(e') = \mathsf{width}(e)$ and $\mathsf{height}(\widetilde{e}) = \mathsf{height}(\widetilde{e'}) = \mathsf{height}(e') = \mathsf{height}(e)$.
- If $e = ||(e_1, \ldots, e_n)$, let $(k_1, \ldots, k_n) \in (\mathbb{N}^+)^n$ and $\left(e'_{i1}, \ldots, e'_{ik_i}\right) \in \mathfrak{E}_*^{k_i}$ for $i \in [\![1, n]\!]$ such that for all $i \in [\![1, n]\!]$, $\widetilde{e_i} = ||\left(e'_{i1}, \ldots, e'_{ik_i}\right)$. Then $\widetilde{e} = ||\left(e'_{11}, \ldots, e'_{nk_n}\right)$.
  For all $i \in [\![1, n]\!]$, we have:

  1. By Definition 22, $\mathsf{width}(\widetilde{e_i}) = \mathsf{width}\left(e'_{i1}\right)$ and $\mathsf{height}(\widetilde{e_i}) = k_i$;
  2. By induction hypothesis, $\mathsf{width}(\widetilde{e_i}) = \mathsf{width}(e_i)$ and $\mathsf{height}(\widetilde{e_i}) = \mathsf{height}(e_i)$.

Therefore,

$$\begin{aligned}
\mathsf{width}(\widetilde{e}) &= \max\left(\mathsf{width}\left(e'_{11}\right), \ldots, \mathsf{width}\left(e'_{nk_n}\right)\right) \\
&= \max_{i=1}^{n} \mathsf{width}\left(e'_{i1}\right) \\
&= \max_{i=1}^{n} \mathsf{width}\left(\widetilde{e_i}\right) \\
&= \mathsf{width}(e); \\
\mathsf{height}(\widetilde{e}) &= \sum_{i=1}^{n} k_i \\
&= \sum_{i=1}^{n} \mathsf{height}\left(\widetilde{e_i}\right) \\
&= \sum_{i=1}^{n} \mathsf{height}\left(e_i\right) \\
&= \mathsf{width}(e).
\end{aligned}$$

□

The other important result that we need to prove is, essentially, that relaxation can only increase the number of states in the IDL graph. Unfortunately, this is not true when all states are taken into account: relaxation sometimes leads to merging interleave branches, resulting in disappearing split and merge nodes. But the failing argument can be fixed by considering those states that actually matter in the complexity analysis, i.e. those from which no ε-transition is available. This is captured by the following definition of so-called *stable states*; states, that have no outgoing ε-transition.

Definition 23 (*Stable states*) Let Σ be a set of symbols that does not contain $\underline{\varepsilon}$ and $\diamond$ and e an IDL expression over Σ that does not contain $\vee$. Let $\mathscr{S}_e$ be the state space of e. The set of *general* stable states of e is defined as

$$\tilde{\mathscr{S}_e} = \left\{s \in \mathscr{S}_e \mid \forall s' \in \mathscr{S}_e, \neg\Delta_e\left(s, \underline{\varepsilon}, \emptyset, s'\right)\right\},$$

the set of *t-specific* stable states of e as

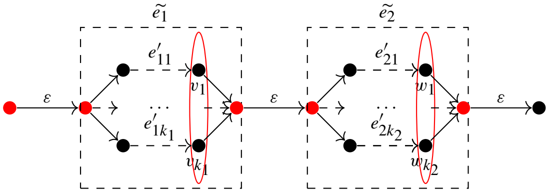
$$\tilde{\mathscr{S}}_{e,t} = \left\{s \in \mathscr{S}_{e,t} \mid \forall s' \in \mathscr{S}_{e,t}, \neg\Delta_e\left(s, \underline{\varepsilon}, \emptyset, s'\right)\right\}.$$

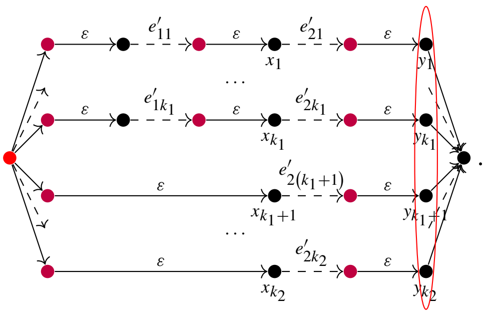
We can now prove that through relaxation, the number of (specific) stable states can only increase.

Proposition 10 *Let Σ be a set of symbols that does not contain $\underline{\varepsilon}$ and $\diamond$ and e an IDL expression over Σ that does not contain $\vee$. For $t \in \Sigma^*$, we have $\left|\tilde{\mathscr{S}}_{e,t}\right| \leq \left|\tilde{\mathscr{S}}_{\tilde{e},t}\right|$.*

Proof We prove this by induction on $e \in \mathfrak{E}$.

- If $e = a$, $\widetilde{e} = e$ and there is nothing to prove.
- If $e = e_1 \cdot e_2$, assume without loss of generality that $k_2 \geq k_1$.
  The situation is as follows (implicit $\vdash_i$ and $\dashv_i$ labels have been omitted):

$\widetilde{e_1} \cdot \widetilde{e_2} =$ 

$\widetilde{e_1 \cdot e_2} =$  .

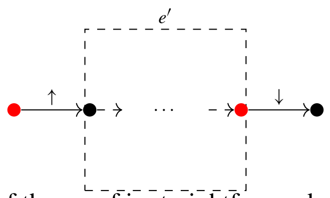
Red vertices and sets denote unstable (non-stable) states. Purple vertices denote vertices that cannot belong to any stable state because they have the left end of an ε-transition.
We first prove that for all $t \in \mathscr{T}$, $\left|\widetilde{\mathscr{S}}_{\widetilde{e_1}\cdot\widetilde{e_2},t}\right| \leq \left|\widetilde{\mathscr{S}}_{\widetilde{e_1\cdot e_2},t}\right|$. This can be done by identifying, for each $t \in \mathscr{T}$, an injective map $\widetilde{\mathscr{S}}_{\widetilde{e_1}\cdot\widetilde{e_2},t} \to \widetilde{\mathscr{S}}_{\widetilde{e_1\cdot e_2},t}$.
Let $t \in \mathscr{T}$. We build a map f as follows:

- Each stable state whose cut is in $\widetilde{e_1}$ is mapped to its natural counterpart in $\widetilde{e_1 \cdot e_2}$, where v_i is replaced by x_i for all $i \in [\![1, k_1]\!]$ and all $(x_i)_{i \in [\![k_1+1, k_2]\!]}$ are added with stack $[\emptyset, \emptyset]$;
- Each stable state whose cut is in $\widetilde{e_2}$ is mapped to its natural counterpart in $\widetilde{e_1 \cdot e_2}$, where w_i is replaced by y_i for all $i \in [\![1, k_2]\!]$;
- The final state of $\widetilde{e_1} \cdot \widetilde{e_2}$ is mapped to the final state of $\widetilde{e_1 \cdot e_2}$.

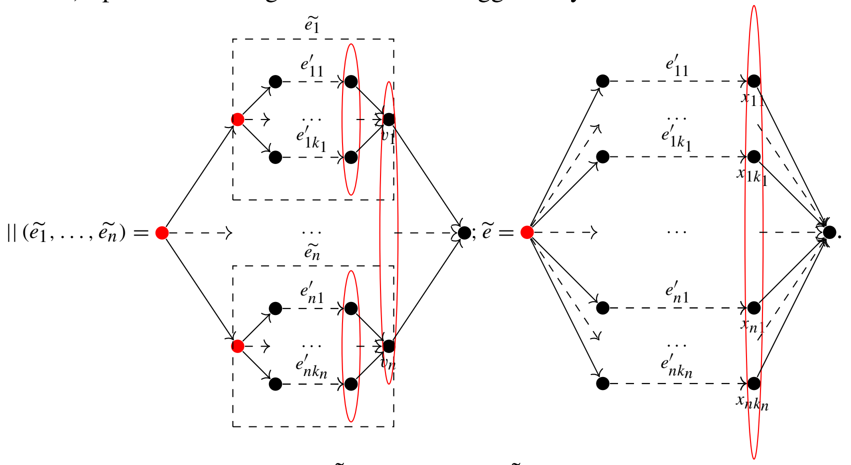
It is easy to see that f is injective. Therefore, $\left|\widetilde{\mathscr{S}}_{\widetilde{e_1}\cdot\widetilde{e_2},t}\right| \leq \left|\widetilde{\mathscr{S}}_{\widetilde{e_1\cdot e_2},t}\right|$.

We then have to check that for all $t \in \mathscr{T}$, $\left|\widetilde{\mathscr{S}}_{e_1\cdot e_2,t}\right| \leq \left|\widetilde{\mathscr{S}}_{\widetilde{e_1}\cdot\widetilde{e_2},t}\right|$.
Let $t \in \mathscr{T}$. We build an injective map f as before. Note that cuts internal to either $\widetilde{e_1}$ or $\widetilde{e_2}$ are easy to handle: the top of the stack is used independently in every branch to decide whether a given state can be accessed or not. The only difficulty thus concerns the final cut. Now, the states provided by the final cut of any IDL expression e' are in bijection with the sets of positions $p \in \mathscr{P}([\![1, |t|]\!])$ such that t_p is matched by e'. Lemma 4 then concludes the proof.
Now, we have

$$\left|\widetilde{\mathscr{S}}_{\widetilde{e},t}\right| = \left|\widetilde{\mathscr{S}}_{\widetilde{e_1\cdot e_2},t}\right| \geq \left|\widetilde{\mathscr{S}}_{\widetilde{e_1}\cdot\widetilde{e_2},t}\right| \geq \left|\widetilde{\mathscr{S}}_{e_1\cdot e_2,t}\right| = \left|\widetilde{\mathscr{S}}_{e,t}\right|.$$

- If $e = \times(e')$, $\times(e') =$ 

(with the same color-coding as above); the rest of the proof is straightforward.
- If $e = \,||\,(e_1, \ldots, e_n)$, the situation is as follows (with the same conventions as before) up to a reordering of the branches triggered by sort:



We build a bijective map $f : \tilde{\mathscr{S}}_{||(\widetilde{e_1},\ldots,\widetilde{e_n}),t} \to \tilde{\mathscr{S}}_{\widetilde{e},t}$ as follows:

- Let $s \in \tilde{\mathscr{S}}_{||(\widetilde{e_1},\ldots,\widetilde{e_n}),t}$. If s is not the final node, write s as $s_1 \otimes \cdots \otimes s_n$ with $s_i \in \tilde{\mathscr{S}}_{\widetilde{e_i},t}$ for $i \in [\![1, n]\!]$. For $i \in [\![1, n]\!]$, define s'_i as the natural counterpart of s_i in $\widetilde{e}$, except for $s_i = \{(v_1, z)\}$ which has no counterpart in $\widetilde{e}$ and is associated with $\left\{(x_{i1}, z_1), \ldots, \left(x_{ik_i}, z_{k_i}\right)\right\}$ where $z_1, \ldots, z_{k_i}$ are stacks chosen to replicate the non-stable state leading to (v_1, z). Put $f(s) = s'_1 \otimes \cdots \otimes s'_n$.
- If s is the final node, map it to the final node of $\widetilde{e}$; associativity of interleave guarantees that the mapping between states based on the final nodes is bijective.

This ensures that $\left|\tilde{\mathscr{S}}_{||(\widetilde{e_1},\ldots,\widetilde{e_n}),t}\right| = \left|\tilde{\mathscr{S}}_{\widetilde{e},t}\right|$. The rest of the proof exploits the same arguments as for concatenation.

□

A.1.2 Counting Parse States in Relaxed IDL Graphs

Notations and assumptions In the remaining part of this subsection, we fix a set of symbols Σ that does not contain $\underline{\varepsilon}$ and $\diamond$ and call $\mathfrak{E}_*$ the set of IDL expressions over Σ that do not contain $\vee$; we further denote by $\tilde{\mathfrak{E}}$ the set of relaxed IDL expressions over Σ. We consider $e \in \tilde{\mathfrak{E}}$ and define $w := \mathsf{width}(e)$, $h := \mathsf{height}(e)$. We choose a set $T \subset \Sigma^*$, an IDL-PMCF grammar G, and put $\rho := \rho_{T,G}$.

We finally choose $t \in T$, let $n := |t|$, $Q_t := \left\{p \subset \mathscr{P}([\![1, |t|]\!]) \mid t_p \in \mathrm{L}(e)\right\}$ and assume $w \geq h$.

The goal of this subsection is to prove Theorem 5 below, that provides an upper bound for $\left|\tilde{\mathscr{S}}_{e,t}\right|$ as a function of w, h and ρ, as well as of the size n of t. The definition of $E_{\ell,n}$ and associated combinatorial results can be found in the appendix.

Proposition 11 *If $h = 1$, we have*

$$|Q_t| \leq \rho^w \left(\frac{n}{w}\right)^w \frac{(n+w-1)^{w-1}}{(w-1)!}.$$

Proof Assume that $h = 1$. For t to be matched by e, which has the form $||\,(a_{11} \cdot \cdots \cdot a_{1w})$, the string t must be decomposed in h (possibly empty) substrings $t_{p_1}, \ldots, t_{p_w}$ such that $p_1 \prec \cdots \prec p_w$, $p = \sqcup_{i=1}^{w} p_i$ and for all $i \in [\![1, w]\!]$, there exists $q_i \subset p_i$ such that $a_{1i} \mathrel{\hat{\rightarrow}} t_{q_i}$. Such p_i are uniquely dermined by their sizes $(k_i := |p_i|)_{i \in [\![1,w]\!]}$. For each such decomposition $(k_1, \ldots, k_w)$, for all $i \in [\![1, w]\!]$, we have

$$\left|\left\{q_i \subset p_i \mid a_{1i} \mathrel{\hat{\rightarrow}} t_{q_i}\right\}\right| \leq \rho\, |p_i| = \rho k_i$$

by definition of ρ. Therefore,

$$\begin{aligned}
|Q_i| &\leq \sum_{k_1+\cdots+k_w=n} \prod_{i=1}^{w} (\rho k_i) \\
&\leq \rho^w \sum_{k_1+\cdots+k_w=n} \left(\frac{\sum_{i=1}^{w} k_i}{w}\right)^w \\
&= \rho^w \left(\frac{n}{w}\right)^w \left|E_{w,n}\right| \\
&\leq \rho^w \left(\frac{n}{w}\right)^w \frac{(n+w-1)^{w-1}}{(w-1)!}.
\end{aligned}$$

□

The following result immediately follows:

Proposition 12 *If $h = 1$, we have*

$$\left|\tilde{\mathscr{S}}_{e,t}\right| \leq 2w\rho^w \left(\frac{n}{w}\right)^w \frac{(n+w-1)^{w-1}}{(w-1)!}.$$

Proof Assume $h = 1$. Given the form of e, the number of cuts of e that can be the base of a stable state is $w + 1$:

$$e = \begin{tikzcd} {\color{red}\bullet} \arrow[r, "\vdash_1"] & {\color{red}\bullet} \arrow[r, "\varepsilon"] & \bullet \arrow[r, "a_{11}"] & {\color{red}\bullet} \arrow[r, "\varepsilon"] & \bullet \arrow[r, "a_{12}"] & {\color{red}\bullet} \arrow[r, dashed] & \cdots \arrow[r, dashed] & \bullet \arrow[r, "a_{1w}"] & {\color{red}\bullet} \arrow[r, "\varepsilon"] & {\color{red}\bullet} \arrow[r, "\dashv_1"] & \bullet \end{tikzcd}.$$

Each of these cuts contains exactly one vertex that can be viewed as the end state of the IDL graph of some $e' \in \hat{\mathfrak{E}}$ such that $e' = \| (a_{11} \cdot \cdots \cdot a_{1k'})$ with $k' \leq w$. As our bound on $|Q_{t'}|, t' \in T$, depends only on $\left|t'\right|$ and w and is decreasing in both variables, there follows:

$$\left|\tilde{\mathscr{S}}_{e,t}\right| \leq (w+1)\,|Q_t| \leq 2w\,|Q_t| \leq 2w\rho^w \left(\frac{n}{w}\right)^w \frac{(n+w-1)^{w-1}}{(w-1)!}.$$

□

Building on the result for $h = 1$, we can handle the general case.

Proposition 13 *We have*

$$|Q_t| \leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \left(\frac{n}{h} + w - 1\right)^{2hw} h^n.$$

Proof For t to be matched by e, which has the form $\| (e_1, \ldots, e_h)$, the string t must be decomposed in w (possibly empty) substrings $t_{p_1}, \ldots, t_{p_h}$ such that $p = \sqcup_{i=1}^h p_i$ and for all $i \in [\![1, h]\!]$, there exists $q_i \subset p_i$ such that $a_{1i} \hat{\rightarrow} t_{q_i}$. For each choice of $(k_i := |p_i|)_{i \in [\![1,h]\!]}$, there are $\binom{n}{k_1,\ldots,k_h}$ possible choices of $(p_i)_{i\in[\![1,h]\!]}$ such that the above condition holds. This yields:

$$\begin{aligned}
|Q_t| &\leq \sum_{k_1+\cdots+k_h=n} \binom{n}{k_1,\ldots,k_h} \prod_{i=1}^h \left[\rho^w \left(\frac{k_i}{w}\right)^w \frac{(k_i+w-1)^{w-1}}{(w-1)!}\right] \\
&\leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \sum_{k_1+\cdots+k_h=n} \binom{n}{k_1,\ldots,k_h} \\
&\quad \times \left(\prod_{i=1}^h (k_i+w-1)\right)^{w-1} \left(\prod_{i=1}^h k_i\right)^w \\
&\leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \left(\frac{n}{h}+w-1\right)^{h(w-1)} \left(\frac{n}{h}\right)^{hw} \sum_{k_1+\cdots+k_h=n} \binom{n}{k_1,\ldots,k_h} \\
&\leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \left(\frac{n}{h}+w-1\right)^{h(w-1)} \left(\frac{n}{h}\right)^{hw} h^n \\
&\leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \left(\frac{n}{h}+w-1\right)^{2hw} h^n.
\end{aligned}$$

□

Theorem 5 *We have*

$$\left|\tilde{\mathscr{S}}_{e,t}\right| \leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \left[\left(2\,(w+1)^2\right)^h + h^n\right] \left(\frac{n}{h}+w\right)^{2hw}.$$

Proof As $e \in \hat{\mathfrak{E}}$, the number of cuts of e that can be the base of a stable state is $|C_e| = (w+1)^h + 2$. Except the start and end cuts s_e and f_e, that contain exactly one vertex, every other cut contains exactly h vertices. The start cut s_e is unstable. The final cut has $|Q_t|$ possible states (its stack has depth 1). For any other cut c, for all $v \in c$, v can be regarded as the end state of the IDL graph of some relaxed IDL expression e' corresponding to the branch of e containing v. All stacks on this branch contain two elements : (i) a head (ii) a head-of-tail that is equal to the empty set. By construction $\mathsf{width}\left(e'\right) \leq \mathsf{width}\,(e) = w$. Therefore, vertex v provides at most $\left|\tilde{\mathscr{S}}_{e',t}\right|$ stack choices. Let S be the supremum of all such $\left|\tilde{\mathscr{S}}_{e',t}\right|$. Finally

$$\begin{aligned}
\left|\tilde{\mathscr{S}}_{e,t}\right| &\leq (w+1)^h\, S^h + |Q_t| \\
&\leq (w+1)^h \left[2w\rho^w \left(\frac{n}{w}\right)^w \frac{(n+w-1)^{w-1}}{(w-1)!}\right]^h \\
&\quad + \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \left(\frac{n}{h} + w - 1\right)^{2hw} h^n \\
&\leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \\
&\quad \times \left[(2w\,(w+1))^h\, n^w\, (n+w-1)^{w-1} + \left(\frac{n}{h} + w - 1\right)^{2hw} h^n\right] \\
&\leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \left[\left(2\,(w+1)^2\right)^h (n+w)^{2w} + \left(\frac{n}{h} + w\right)^{2hw} h^n\right].
\end{aligned}$$

As $w \geq h$,

$$\begin{aligned}
\left(\frac{n}{h} + w\right)^h &= \frac{(n+hw)^h}{h^h} = \frac{(n+hw)^{h-2}}{h^{h-2}} \frac{(n+hw)^2}{h^2} \\
&\geq \frac{(n+hw)^2}{h^2} = \frac{n^2}{h^2} + \underbrace{\frac{2nw}{h}}_{\geq n} + \underbrace{w^2}_{\geq w} \geq n + w;
\end{aligned}$$

hence

$$\left|\tilde{\mathscr{S}}_{e,t}\right| \leq \left(\left(\frac{\rho}{w}\right)^w \frac{1}{(w-1)!}\right)^h \left[\left(2\,(w+1)^2\right)^h + h^n\right] \left(\frac{n}{h} + w\right)^{2hw}.$$

□

A.1.3 Final Complexity Estimate

We now discuss the complexity of each of the functions presented in Algorithm 2 in the case where none of the IDL expressions used in the grammar uses any $\vee$ node.

As every IDL-PMCF grammar can be easily translated into an equivalent IDL-PMCFG without $\vee$ node by adding new rules, this does not restrict the set of languages that can be matched. At the end of this section, we will briefly address the role of the $\vee$ node in the complexity of our algorithm and propose a simple approach to efficiently measuring it.

Complexity of the algorithm is measured in terms of number of elementary operations. Creating and adding an item to the current environment is considered a $\mathcal{O}(1)$. Listing available transitions to new states given a state, a (non)terminal and an associated set of positions is an $\mathcal{O}(hn)$: the size of a cut is at most h and operations on set positions have a complexity of $\mathcal{O}(n)$. The constant is improved in practice by pre-computing all available transitions for every new active item to be added to the environment. Denoting by α the maximal arity of any rule in G ($G := \max_{f \in F} a(f)$), we find that reserve is an $\mathcal{O}(\alpha m)$, whereas compat and unify are $\mathcal{O}(\alpha mn)$. Initializing a context is also an $\mathcal{O}(\alpha m)$.

To simplify the analysis, the contribution of any of the 8 mutually recursive functions is evaluated by multiplying an upper bound of the total number of times it will be called by the cost of the elementary operations it uses, excluding recursive calls. The number of rules in G is defined as $R = |P|$ and the maximal number of nodes in any IDL expression in G is denoted by M. We will show that the number of times each function is called can be easily bounded by a function of the maximal (or, equivalently, final) number of:

- Active items in E, which we denote by A;
- Passive items in E, which we denote by P;
- Complete items in E, which we denote by C.

We can now inspect the complexity of each of the nine functions presented in Algorithm 2.

Parse The two `for` loops have a complexity of $\mathcal{O}(nA)$.

PredictAll The function itself is called only once by Parse. The internal code is executed mR times at most. The internal code consists of a context initialization ($\mathcal{O}(\alpha m)$), and an item creation ($\mathcal{O}(1)$). The contribution of PredictAll is therefore an $\mathcal{O}\left(m^2 R\alpha\right)$.

TryScan The function is called at most $\mathcal{O}(nA)$ times by Parse. Computing available transitions is an $\mathcal{O}(hn)$ (see above). For each transition, the cost of operations is constant. This results in an overall $\mathcal{O}\left(Ahn^2\right)$.

TrySave The function is called exactly P times. After a constant-cost test and identification of the production at stake, new items are added to the environment in time $\mathcal{O}(1)$, resulting in an $\mathcal{O}(P)$.

TryStep The function is called at most $mAMR$ times: each node is processed at most once for each new active item. Its cost is an $\mathcal{O}(hn)$. Its overall contribution to complexity is therefore $\mathcal{O}(hmAMRn)$.

TryCombineLHS The function is called exactly A times. Each of the $\mathcal{O}(C)$ iterations of the outer loop causes $\mathcal{O}(hn)$ iterations of the inner loop, each of which has cost $\mathcal{O}(\alpha mn)$. This yields an $\mathcal{O}\left(AC\alpha hmn^2\right)$.

TryCombineRHS The function is called exactly C times. Each of the $\mathcal{O}(A)$ iterations of the outer loop costs $\mathcal{O}\left(\alpha hmn^2\right)$ as above. This yields another $\mathcal{O}\left(AC\alpha hmn^2\right)$.

TryUnify The function is called exactly P times. The total number of complete items of a given category in the environment is $(\rho n+1)^m$ (each field provides at most ρn matches from which at most one must be selected). At most that many items must be checked by the outer loop; the inner loop then has cost $\mathcal{O}(\alpha mn)$. The final complexity is $\mathcal{O}\left(P\alpha mn\,(\rho n)^m\right)$.
The total complexity of the algorithm is now

$$\begin{aligned}
&\mathcal{O}\left(nA + m^2R\alpha + Ahn^2 + P + hmAMRn + AC\alpha hmn^2 + P\alpha mn\,(\rho n)^m\right)\\
&\quad = \mathcal{O}\left(m^2R\alpha + hmAMRn + AC\alpha hmn^2 + P\alpha mn\,(\rho n)^m\right)\\
&\quad = \mathcal{O}\left(m\left(mR\alpha + AMRh^2n + AC\alpha hn^2 + P\alpha n\,(\rho n)^m\right)\right).
\end{aligned}$$

Using the results of the previous part and the remarks above, we can now complete simple upper bounds for A, C and P as follows:

- $P = \mathcal{O}(\rho n)$;
- $C = \mathcal{O}\left((\rho n)^m\right)$;
- $A = \mathcal{O}(RmS)$ where S is an upper bound of all $\left|\tilde{\mathscr{S}}_{e,t}\right|$ for e in G.

Our final theorem follows:

Theorem 4 *Let $m \in \mathbb{N}$. Let $G = (N, \delta, \Sigma, F, P, S)$ be an m-parallel IDL-MCF grammar. Let E be the set of IDL expressions used in G. Assume that for all $e \in E$, e does not contain any $\vee$ node. Let $T \subset \Sigma^*$ and put $\rho := \rho_T$. Let $w := \max_{e\in E}$ width(e), $h := \max_{e\in E}$ height(e), $R = |P|$, $\alpha = \max_{f\in F} a(f)$, $M = \max_{e\in E}|e|$. Finally, let $t \in T$ and $n := |t|$; let A, P and C as above and $S = \max_{e\in E}\left|\tilde{\mathscr{S}}_{e,t}\right|$. Assume that $w \geq 6$ and $n \geq w$. An upper bound of the complexity of the parsing algorithm described in Algorithm 2 is given by*

$$\mathcal{O}\left(\alpha m^2Rh\rho^m\left(\left(\frac{\rho w^2}{h^2}\right)^w\frac{1}{(w-1)!}\right)^h h^n n^{2hw+m+2}\right).$$

Proof According to the above discussion, the complexity of parsing t with grammar G, denoted by $c(G,t)$, is such that

$$\begin{aligned}
c(G,t) &= \mathcal{O}\left(m\left(mR\alpha + AMRh^2n + AC\alpha hn^2 + P\alpha n\,(\rho n)^m\right)\right)\\
&= \mathcal{O}\left(m\left(\alpha n\,(\rho n)^{m+1} + \alpha SmRhn^2\,(\rho n)^m\right)\right).
\end{aligned}$$

Let E' be the image of E by $e \mapsto \tilde{e}$ and $S' = \max_{e'\in E'}\left|\tilde{\mathscr{S}}_{e',t}\right|$.

Let $e \in E$, $\hat{w} := \mathsf{width}\,(e)$, $\hat{h} := \mathsf{height}\,(e)$, $w' := \mathsf{width}\,(\widetilde{e})$, $h' := \mathsf{height}\,(\widetilde{e})$. By Proposition 9, $w' = \hat{w} \leq w$ and $h' = \hat{h} \leq h$. According to Proposition 10, $\left|\tilde{\mathscr{S}}_{e,t}\right| \leq \left|\tilde{\mathscr{S}}_{\widetilde{e},t}\right|$ and by Theorem 5,

$$\left|\tilde{\mathscr{S}}_{\widetilde{e},t}\right| \leq \left(\left(\frac{\rho}{w}\right)^{w} \frac{1}{(w-1)!}\right)^{h} \left[\left(2\,(w+1)^2\right)^{h} + h^n\right] \left(\frac{n}{h} + w\right)^{2hw}.$$

Hence

$$S \leq \left(\left(\frac{\rho}{w}\right)^{w} \frac{1}{(w-1)!}\right)^{h} \left[\left(2\,(w+1)^2\right)^{h} + h^n\right] \left(\frac{n}{h} + w\right)^{2hw}$$

and

$$\begin{aligned}
c\,(G,t) &= \mathscr{O}\Bigg(m\Bigg(\alpha n\,(\rho n)^{m+1} \\
&\quad + \left\{\left(\left(\frac{\rho}{w}\right)^{w} \frac{1}{(w-1)!}\right)^{h} \left[\left(2\,(w+1)^2\right)^{h} + h^n\right] \left(\frac{n}{h} + w\right)^{2hw}\right\} \\
&\quad \times \alpha m R h n^2\,(\rho n)^m\Bigg)\Bigg) \\
&= \mathscr{O}\Bigg(\left[\left(\left(\frac{\rho}{w}\right)^{w} \frac{1}{(w-1)!}\right)^{h} \left[\left(2\,(w+1)^2\right)^{h} + h^n\right] \left(\frac{n}{h} + w\right)^{2hw}\right] \\
&\quad \times \alpha m^2 R h n^2\,(\rho n)^m\Bigg) \\
&= \mathscr{O}\left(\left(\left(\frac{\rho}{w}\right)^{w} \frac{1}{(w-1)!}\right)^{h} h^n \left(\frac{n}{h} + w\right)^{2hw} \alpha m^2 R h n^2\,(\rho n)^m\right) \\
&= \mathscr{O}\left(\alpha m^2 R h \rho^m \left(\left(\frac{\rho w^2}{h^2}\right)^{w} \frac{1}{(w-1)!}\right)^{h} h^n n^{2hw+m+2}\right)
\end{aligned}$$

□

When some IDL expressions found in the grammar are making use of the $\vee$ node, it is always possible to rewrite it to produce an equivalent grammar with more rules that does not contain any $\vee$ node. It seems reasonable to conjecture that the parsing complexity with the rewritten grammar can only be worse than with the original grammar. In this case, the complexity of parsing the original grammar can be bounded by V times the above complexity, while V is an upper bound of the number of new rules produced for each old rule in the grammar.

A.2 A Few Combinatorial Results

We thank Pierre-Alain Sallard for suggesting the expression of Lemma 7.

For $(\ell, n) \in \mathbb{N}^2$, define

$$E_{\ell,n} = \left\{(k_1, \dots, k_\ell) \in \mathbb{N}^\ell \mid k_1 + \cdots + k_\ell = n\right\}$$
$$F_{\ell,n} = \left\{(k_1, \dots, k_\ell) \in \mathbb{N}^\ell \mid k_1 + \cdots + k_\ell \leq n\right\}.$$

Lemma 5 *For* $(\ell, n) \in \mathbb{N}^+ \times \mathbb{N}$, $\left|E_{\ell,n}\right| = \left|F_{\ell-1,n}\right|$.

Proof Let $(\ell, n) \in \mathbb{N}^+ \times \mathbb{N}$. Let

$$\Phi : E_{\ell,n} \to F_{\ell-1,n}, (k_1, \dots, k_\ell) \mapsto (k_1, \dots, k_{\ell-1})$$
$$\Psi : F_{\ell-1,n} \to E_{\ell,n}, (k_1, \dots, k_{\ell-1}) \mapsto \left(k_1, \dots, k_{\ell-1}, n - \sum_{i=1}^{\ell-1} k_i\right).$$

We have $\Phi \circ \Psi = \mathrm{id}_{E_{\ell,n}}$, $\Psi \circ \Phi = \mathrm{id}_{F_{\ell-1,n}}$; Φ and Ψ are therefore reciprocal bijections, which concludes the proof. $\square$

Lemma 6 *For* $(\ell, n) \in \mathbb{N}^2$. *We have*

$$\left|F_{\ell,n}\right| = \binom{n+\ell}{\ell}.$$

Proof Let $(\ell, n) \in \mathbb{N}^2$. Define predicates $(A_i)_{i\in[\![1,\ell]\!]}$ and $\left(B_j\right)_{j\in[\![1,n]\!]}$ as follows:

$$\forall i \in [\![1, \ell]\!],\ A_i : F_{\ell,n} \to \{\top, \bot\}, k \mapsto k_i = 0$$
$$\forall p \in [\![1, n]\!],\ B_p : F_{\ell,n} \to \{\top, \bot\}, \exists m \in [\![1, \ell]\!],\ \sum_{i=1}^{m} k_i = p.$$

Now, let $\omega = \bigcup_{i=1}^{\ell} A_i \cup \bigcup_{j=1}^{n} B_j$ and $\Omega = \left\{\alpha \in \omega^\ell \mid \forall (\beta_1, \beta_2) \in \alpha^2,\ \beta_1 \neq \beta_2\right\}$.

Clearly, $|\omega| = n + \ell$ and $|\Omega| = \binom{n+\ell}{\ell}$. We now prove that $|\Omega| = \left|F_{\ell,n}\right|$.

More precisely, be prove that for all $\alpha \in \Omega$, there exists one and only one $k \in F_{\ell,n}$ such that $\forall \beta \in \alpha,\ \beta(k) = \top$.

Let $\alpha \in \Omega$ and $\delta = \alpha \cap \left(B_p\right)_{p\in[\![1,n]\!]}$. Let $I = \{i \in [\![1, \ell]\!] \mid A_i \in \alpha\}$ and $J = [\![1, \ell]\!]\backslash I$.

We build a $k \in \mathbb{N}^\ell$ as follows:

- For all $i \in I$, we set $k_i := 0$;
- Let $J =: \left\{i_1, \dots, i_{|\delta|}\right\}$ such that $i_1 < \cdots < i_{|\delta|}$;
- Let $\delta =: \left\{B_{p_1}, \dots, B_{p_{|\delta|}}\right\}$ such that $p_1 < \cdots < p_{|\delta|}$;
- Set $k_{i_1} := p_1$;
- For $j \in [\![2, |\delta|]\!]$, set $k_{i_j} := p_j - p_{j-1}$.

We have

$$\begin{aligned}\sum_{i=1}^{\ell} k_i &= \sum_{i\in I} k_i + \sum_{i\in J} k_i \\ &= \sum_{i\in I} 0 + \sum_{j\in[\![1,|\delta|]\!]} k_{i_j} \\ &= p_1 + \sum_{j\in[\![2,|\delta|]\!]} \left(p_j - p_{j-1}\right) \\ &= p_{|\delta|} \le n;\end{aligned}$$

hence, $k \in F_{\ell,n}$.

Now, let $k \in F_{\ell,n}$, $(\alpha_1, \alpha_2) \in \Omega^2$ such that the above procedure applied to α_1 and α_2 produces the same output k. Let δ_1, δ_2, I_1, I_2, J_1, J_2 as above. First, note that for $q \in \{1, 2\}$, $i \in [\![1, \ell]\!]$, $A_i \in \alpha_q \Leftrightarrow k_i = 0$. This means that for $i \in [\![1, \ell]\!]$, $A_i \in \alpha_1 \Leftrightarrow A_i = \alpha_2$, i.e. $I_1 = I_2$ and therefore $J_1 = J_2$. Let $J_1 = J_2 =: \left\{i_1, \ldots, i_{|\delta_1|}\right\}$ such that $i_1 < \cdots < i_{|\delta_1|}$. Both α_1 and α_2 contain $\gamma := |\delta_1| = |\delta_2|$ elements of $\left(B_j\right)_{j\in[\![1,n]\!]}$. Let $\left(m_1^1, \ldots, m_1^\gamma\right) \in [\![1, \ell]\!]^\gamma$ and $\left(m_2^1, \ldots, m_2^\gamma\right) \in [\![1, \ell]\!]^\gamma$ such that $\delta_1 = \left\{B_{\sum_{i=1}^{m_1^j} k_i} \mid j \in [\![1, \gamma]\!]\right\}$ and $\delta_2 = \left\{B_{\sum_{i=1}^{m_2^j} k_i} \mid j \in [\![1, \gamma]\!]\right\}$. Without loss of generality, we can assume $m_1^j \in J_1$ and $m_2^j \in J_2 = J_1$ for all $j \in [\![1, \gamma]\!]$. As $|\delta_1| = |\delta_2| = |J_1|$, we have $\delta_1 = \delta_2$, and finally $\alpha_1 = \alpha_2$.

Therefore, for all $\alpha \in \Omega$, there exists one and only one $k \in F_{\ell,n}$ such that $\forall\beta \in \alpha$, $\beta(k) = \top$, which concludes the proof. □

Lemma 7 *For* $(\ell, n) \in \mathbb{N}^2$, $\left|E_{\ell,n}\right| = \begin{cases} 0 & \text{if } \ell = 0 \\ \binom{n+\ell-1}{\ell-1} & \text{otherwise} \end{cases}$ *and if* $\ell \ge 1$, $\left|E_{\ell,n}\right| \le \frac{(n+\ell-1)^{\ell-1}}{(\ell-1)!}$.

Proof For $n \in \mathbb{N}$, $\left|E_{\ell,n}\right| = |\emptyset| = 0$.

For $(\ell, n) \in \mathbb{N}^+ \times \mathbb{N}$, we have $\left|E_{\ell,n}\right| = \left|F_{\ell-1,n}\right|$ by Lemma 5 and $\left|F_{\ell-1,n}\right| = \binom{n+\ell-1}{\ell-1}$ by Lemma 6. Finally,

$$\left|E_{\ell,n}\right| = \frac{(n+\ell-1)!}{n!\,(\ell-1)!} \le \frac{(n+\ell-1)^{\ell-1}}{(\ell-1)!}.$$

□

References

Aho, A. V. (1968). Indexed grammars—an extension of context-free grammars. *Journal of the ACM, 15*(4), 647–671. https://doi.org/10.1145/321479.321488.

Angelov, K. (2009). Incremental parsing with parallel multiple context-free grammars. In *Proceedings of the 12th conference of the European chapter of the ACL (EACL 2009)* (pp. 69–76).

Angelov, K., Bringert, B., & Ranta, A. (2009). PGF: A portable run-time format for type-theoretical grammars. *Journal of Logic, Language and Information, 19*(2), 201–228. https://doi.org/10.1007/s10849-009-9112-y.

Austin, P. (2001). Word order in a free word order language: The case of Jiwarli. *Forty Years on: Ken Hale and Australian Languages*, 205–323.

Balusu, R. (2006). Distributive reduplication in Telegu. In *NELS* (Vol. 36).

Becker, T., Rambow, O., & Niv, M. (1992). *The derivational generative power of formal systems or scrambling is beyond LCFRS*. University of Pennsylvania, Institute for Research in Cognitive Science.

Bharati, A., & Sangal, R. (1993). Parsing free word order languages in the Paninian framework. In *Proceedings of the 31st annual meeting on association for computational linguistics* (pp. 105–111). Association for Computational Linguistics. https://doi.org/10.3115/981574.981589.

Bortolussi, B. (2006). Ordre des mots et thématisation en latin. *Linx*, 33–47. https://doi.org/10.4000/linx.378.

Boullier, P. (1998). *Proposal for a natural language processing syntactic backbone*. PhD thesis, INRIA.

Chomsky, N. (1956). Three models for the description of language. *IEEE Transactions on Information Theory, 2*(3), 113–124. https://doi.org/10.1109/tit.1956.1056813.

Conrad, C. (1965). Traditional patterns of word-order in Latin epic from Ennius to Vergil. *Harvard studies in Classical Philology, 69,* 195–258. https://doi.org/10.2307/310783.

Covington, M. A. (1990). A dependency parser for variable-word-order languages. Technical Report AI 1990 01, The University of Georgia.

Dalrymple, M., & Mofu, S. (2011). Plural semantics, reduplication, and numeral modification in Indonesian. *Journal of Semantics, 29*(2), 229–260. https://doi.org/10.1093/jos/ffr015.

Danckaert, J. M. L. (2017). *The development of Latin clause structure: A study of the extended verb phrase* (Vol. 24). Oxford University Press.

Devine, A. M., & Stephens, L. D. (2006). *Latin word order*. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780195181685.001.0001.

Ho, M.-C. (2018). The word problem of Zn is a multiple context-free language. *Groups Complexity Cryptology, 10*(1), 9–15. https://doi.org/10.1515/gcc-2018-0003.

Hoffman, B. A. (1995). The computational analysis of the syntax and interpretation of “free” word order in Turkish. Technical Report 130, IRCS.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2013). *Introduction to automata theory, languages, and computation: Pearson new international edition*. Pearson Education Limited. ISBN 1292039051.

Johnson, S. C. et al. (1975). *Yacc: Yet another compiler–compiler* (Vol. 32). Bell Laboratories.

Kallmeyer, L. (2010). *Parsing beyond context-free grammars*. Springer. https://doi.org/10.1007/978-3-642-14846-0.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations* (pp. 85–103). Springer US. https://doi.org/10.1007/978-1-4684-2001-2_9.

Kashket, M. B. (1986). Parsing Warlpiri—a free word order language. In *Proceedings of the 24th annual meeting on association for computational linguistics* (pp. 60–66). Association for Computational Linguistics, Springer Netherlands. https://doi.org/10.1007/978-94-011-3474-3_5.

Kay, M., & Karttunen, L.. (1984). Parsing in a free word order language. *Natural language parsing, Psychological, computational, and Theoretical Prespectives*, 279–306.

Kirman, J., & Salvati, S. (2013). On the complexity of free word orders. In *Formal grammar* (pp. 209–224). Springer.

Kiss, K. E. (1981). Structural relations in Hungarian, a “free” word order language. *Linguistic Inquiry, 12*(2), 185–213.

Kobele, G. M. (2006). *Generating copies: An investigation into structural identity in language and grammar*. PhD thesis, University of California, Los Angeles.

Koch, U. (1993). The enhancement of a dependency parser for Latin. *Artificial Intelligence*.

Koster, C. H. A. (1991). Affix grammars for natural languages. In *International summer school on attribute grammars, applications, and systems* (pp. 469–484). Springer. https://doi.org/10.1007/3-540-54572-7_19.

Lange, H. (2017). Implementation of a Latin grammar in grammatical framework. In *DATeCH* (pp. 97–102). ACM Press. https://doi.org/10.1145/3078081.3078108.

Ljunglöf, P. (2004). *Expressivity and complexity of the grammatical framework*. PhD thesis, Göteborg University and Chalmers University of Technology, Gothenburg, Sweden.

Ljunglöf, P. (2005). A polynomial time extension of parallel multiple context-free grammar. In *Logical aspects of computational linguistics* (pp. 177–188). Springer. https://doi.org/10.1007/11422532_12.

Ljunglöf, P. (2012). Practical parsing of parallel multiple context-free grammars. In *Proceedings of the 11th international workshop on tree adjoining grammars and related formalisms (TAG+ 11)* (pp. 144–152).

Lötscher, A. (1993). Zur Genese der Verbverdopplung bei gaa, choo, laa, aafaa („gehen", „kommen", „lassen", „anfangen") im Schweizerdeutschen. *Linguistische Berichte Sonderheft, 5,* 180–200. https://doi.org/10.1007/978-3-322-97032-9_9.

Makoto, K., & Salvati, S. (2012). MIX Is not a tree-adjoining language. In *ACL 2012* (pp. 666–674). South Korea.

Marouzeau, J. (1922). *L'ordre des mots dans la phrase latine. 1. Le groupe nominal* (Vol. 1). Champion.

Marouzeau, J. (1949). *L'ordre des mots dans la phrase latine. 3. Les articulations de l'énoncé*. Société d'édition des Belles Lettres.

Martin-Löf, P., & Sambin, G. (1984). *Intuitionistic type theory* (Vol. 9). Bibliopolis Naples. ISBN 8870881059.

Nederhof, M.-J., & Satta, G. (2004). IDL-Expressions: A formalism for representing and parsing finite languages in natural language processing. *Journal of Artificial Intelligence Research, 21,* 287–317. https://doi.org/10.1613/jair.1309.

Pottier, F., & Régis-Gianas, Y. (2016). Menhir reference manual. Technical report.

Pullum, G. K. (1982). Free word order and phrase structure rules. In J. Pustejovsky, & P. Sells (Eds.), *Proceedings of the twelfth annual meeting of the North Eastern linguistic society* (pp. 209–220). University of Massachusetts, Graduate Linguistics Student Association.

Ranta, A. (2004). Grammatical framework. *Journal of Functional Programming, 14*(2), 145–189. https://doi.org/10.1017/S0956796803004738.

Ranta, A. (2011). *Grammatical framework*. Centre for the Study of Language & Information. ISBN 1575866269.

Ranta, A., El Dada, A., & Khegai, J. (2009). The GF resource grammar library. *Linguistic Issues in Language Technology, 2*(2), 1–63.

Reape, M. (1994). *A formal theory of word order: A case study in West Germanic*. PhD thesis, University of Edinburgh.

Salvati, S. (2015). MIX is a 2-MCFL and the word problem in Z2 is captured by the IO and the OI hierarchies. *Journal of Computer and System Sciences, 81*(7), 1252–1277. https://doi.org/10.1016/j.jcss.2015.03.004.

Schaufele, S. W. (1991). *Free word-order syntax: The challenge from Vedic Sanskrit to contemporary formal syntactic theory*. PhD thesis, University of Illinois at Urbana-Champaign.

Seki, H., Matsumura, T., Fujii, M., & Kasami, T. (1991). On multiple context-free grammars. *Theoretical Computer Science, 88*(2), 191–229. https://doi.org/10.1016/0304-3975(91)90374-b.

Shieber, S. M. (1984). Direct parsing of ID/LP grammars. *Linguistics and Philosophy, 7*(2), 135–154. https://doi.org/10.1007/BF00630810.

Siewierska, A., & Uhlirova, L. (1998). An overview of word order in Slavic languages. In A. Siewierska (Ed.), *Constituent order in the languages of Europe* (Vol. 20). Moyton de Grouter. https://doi.org/10.1515/9783110812206.105.

Spevak, O. (2010). *Constituent order in classical Latin prose*. John Benjamins Publishing. https://doi.org/10.1075/slcs.117.

Vijayashanker, K., & Joshi, A. K. (1988). *A study of tree adjoining grammars*. PhD thesis, University of Pennsylvania, Philadelphia.