Word Ordering as a Graph Rewriting Process

Sylvain Kahane[1] and François Lareau[2]

[1] Modyco, Université Paris Ouest—Nanterre La Défense
sylvain@kahane.fr
[2] OLST, Université de Montréal
francois.lareau@umontreal.ca

Abstract. This paper shows how the correspondence between a unordered dependency tree and a sentence that expresses it can be achieved by transforming the tree into a string where each linear precedence link corresponds to one specific syntactic relation. We propose a formal grammar with a distributed architecture that can be used for both synthesis and analysis. We argue for the introduction of a topological tree as an intermediate step between dependency syntax and word order.

Keywords: dependency grammar • linearization • syntax • polarized unification grammar

1 Introduction

Word ordering has been addressed in many formalisms, from Categorial Grammar (CG) and Context Free Grammar (CFG), to Tree-Adjoining Grammar (TAG), Lexical Functional Grammar (LFG), Head-driven Phrase Structure Grammar (HPSG), Minimalist Program (MP), etc. The early formalisms did not separate linearization from sub-categorization. The first formalism to clearly separate them was Generalized Phrase Structure Grammar (GPSG) [1] [2]. In grammars that separate linearization rules from the others, the former are generally expressed in a different formalism than the latter, and the two kinds cannot combine freely with one another. Linearization rules must be precompiled with sub-categorization rules, as in metagrammars for TAG [3], or they form a separate module that is applied as a whole before or after other modules. In LFG, the linearization rules are CFG rules endowed with functional features describing the correspondence between c- and f-structures [4], which cannot be used in other modules of the model [5]. In a constraint-based formalism like HPSG, linear order is constrained by features, but its computation is left aside. It is a list of linear precedence statements from which linear order must be deduced by external mechanisms [6] [7] [8]. More generally, in phrase structure grammar, word order is expressed on constituents but not words: only sister constituents are ordered, and order between words must be deduced by extra devices.

Our first aim in this paper is to propose a general formalism that allows to write both linearization and sub-categorization rules and to combine them in whatever order, with no implicit procedure. This will allow us to use the rules for analysis as well as synthesis, to pre-compile sets of rules from different levels, and to use incremental strategies for parsing or generation. It is important to underline that our approach is mathematical and not computational: our aim is to propose a rule-based formalism to write modular, declarative grammars, but we are not directly interested in procedural strategies.

Our second aim is to propose a formalism for dependency grammar (henceforth DG). In formal DG, the focus has been on valency, sub-categorization, dependency tree generation, and the semantics-syntax interface. One of the well-known advantages of DG is that it separates linear order from syntactic structure proper [9] and it elegantly captures non-projective syntactic constructions (equivalent to trees with discontinuous constituents in phrase-based formalisms). Nevertheless, few efforts have been made on the formalization of word ordering and non-projectivity in DG. Formal DGs handling non-projective ordering have been proposed [10] [11] [12], but they do not fit our first aim, which is to use the same formalism for linearization and sub-categorization rules. Non-projective dependency parsers have been proposed [13], but the grammar cannot be clearly separated from the parsing procedure. Meaning-Text Theory (MTT) [14] is a model we want to follow because it is very modular and all rules are expressed in similar terms, that is, in terms of correspondence between two levels of representation. But although linearization rules have been proposed within MTT [15] [16], a complete formalization has never been achieved, especially the treatment of non-projective ordering, which remains particularly informal. This paper can be viewed as an attempt to give a clean formalization of MTT's linearization rules.

Our third aim is to show that language modelling, and in particular word ordering, can be viewed as a graph rewriting process. We consider that modelling a natural language consists in associating every uttered string of words to its meaning(s), or, conversely, to associate every linguistic meaning to the string(s) of words that express it. Our approach to language modelling is strongly influenced by MTT, which posits that the core meaning of a sentence can be encoded by a graph representing the predicate-argument relations that exist between the meaning of the words (or morphemes) composing it [17]. In other words, language modelling mainly consists in transforming a semantic graph into a linear graph representing the chain of words (and vice versa). Moreover, except where there is no one-to-one correspondence between minimal semantic units and minimal expressive units, most linear precedence links between two consecutive words in the spoken chain are the image of particular predicate-argument relations. Hence, the correspondence between a semantic graph and a string of words mainly consists in moving the edges of the graph until they form a chain.

2 Natural language modelling and graph rewriting

Let us consider the following sentence:

(1) *Men from Swaziland often work in mines.*

The text of (1) is a string of words, which can be represented by a linear graph as in **Fig. 1**. This graph will be called *linear order* in this paper (following [9], [18]). It is similar to the morphological structure of MTT, an intermediate level of representation between the surface syntactic structure and the phonological representation. Edges of the linear order are called *linear precedence links* (LPLs) and are labelled with the symbol “<”.

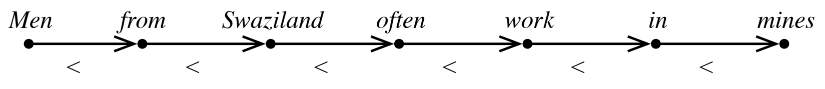


Fig. 1. Linear order of (1)

The core meaning of (1) can be represented as a *semantic graph* as in **Fig. 2** [14], [17]. In a semantic graph, edges are called semantic dependencies and represent predicate-argument relations. The source of a semantic dependency is a predicate, and its target is an argument of that predicate. Edges are labelled with a number indicating the rank of the argument (1st argument, 2nd argument, etc.), in decreasing order of salience [19]. The semantic graph of (1) below expresses the fact that the signified of *work* is a unary predicate taking the signified of *men* as its first (and unique) argument. The signified of *often* is also a unary predicate: it takes the signified of *work* as its argument. The signifieds of both locative prepositions *from* and *in* are binary predicates taking the located entity or event as their first argument and the locus as their second argument. It is also possible to encode this graph with an equivalent logical formula, as below [20] [21].

$$\left[\begin{array}{l} \text{x: men} \\ \text{y: Swaziland} \\ \text{z: mines} \\ \text{e: work(x)} \\ \text{often(e)} \\ \text{from(x,y)} \\ \text{in(e,z)} \end{array}\right] \equiv$$

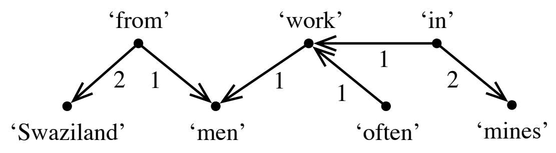


Fig. 2. Semantic representation of (1)

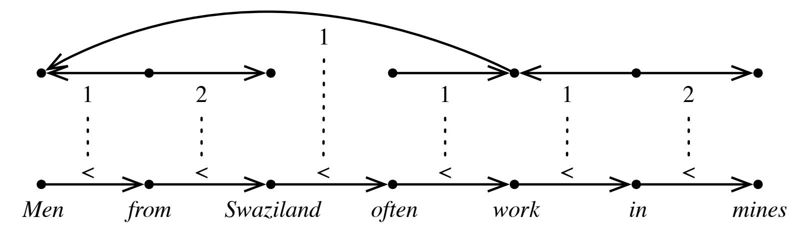


Fig. 3. Correspondence between linear precedence links and semantic dependencies

Fig. 3 shows the correspondence between the LPLs and the semantic dependencies of (1). The correspondence is generally realized in two main stages: 1) a semantics-syntax interface ensuring the lexicalization and the hierarchization of the semantic graph, giving a syntactic dependency tree and 2) the linearization and the morphologization of the dependency tree, giving a string of words. This paper focuses on linearization. The semantics-syntax interface has been described in several papers [14] [22] [23] [24] [25] [26] and will be sketched in Section 3.4. The dependency tree of (1) is presented in **Fig. 4**.

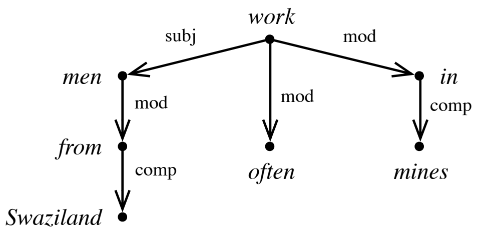


Fig. 4. Syntactic tree of (1)

A syntactic dependency tree for an *n* word sentence has exactly *n*–1 edges, as does its linear order. We will show that the edges are in a one-to-one correspondence and that this correspondence can be described by a graph rewriting system that moves the edges of the dependency tree to transform it into a string. The rules will be written in a formalism called Polarized Unification Grammar (PUG), which was introduced by [27], [28], based on previous work by [29] [30] [31] [32]. It has been used for the semantics-syntax interface since [23] but has never been really used for word ordering before.[1] There are great advantages to using the same formalism for different modules of the grammar. A common formalism allows us to use the same grammar both for synthesis (producing a string of words from a semantic representation) and analysis (extracting the meaning of a sentence). Many strategies are conceivable, including the possibility to pre-compile some groups of rules, the result of such pre-compilation being expressed in the common formalism.

3 Governor-dependent linearization rules

3.1 Sketch of governor-dependent linearization rules

We will introduce our approach with a minimalistic example:

(2) *Mary loves Peter*

Fig. 5 shows the dependency tree for this sentence, where *Mary* is the subject of the verb and *Peter* its object, and the corresponding linear order.

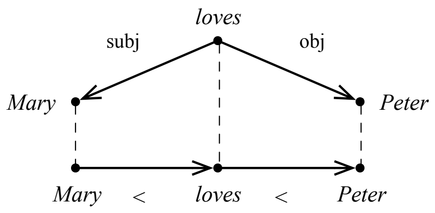


Fig. 5. Syntactic tree and linear order of (2)

To linearize the dependency tree of (2), we only need to know that in English the subject goes before the verb and the object can goes after.[2] This can be formalized by saying that a *subject* dependency corresponds to an LPL in the opposite direction, while the *object* dependency corresponds to an LPL in the same direction.[3] **Fig. 6** gives a first sketch of these rules, before introducing our formal framework.

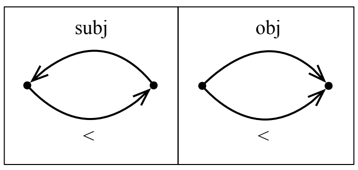


Fig. 6. Sketch of *subj* and *obj* governor-dependent linearization rules

[1] [27,28] showed how to simulate in PUG LFG's phi-projection, which ensures the linearization process. The resulting grammar has not been studied in itself, although a similar grammar was presented in [23].

[2] Other rules can apply to an *object* dependency under particular conditions: for instance a wh-word in the object position can be placed before the verb, but even for a wh-word it is possible to place it after the verb (cf. so called *in situ* wh-question: *Mary loves who?*).

[3] Every rule must be read with an epistemic modality: things *can* happen as per the rule. It comes from the fact that alternative rules can *a priori* apply and things can happen differently.

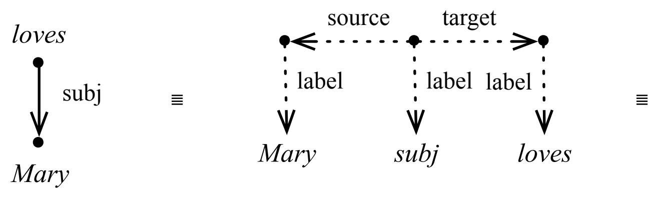
Such linearization rules can be used in synthesis (syntactic dependencies become LPLs) as well as in analysis (LPLs become syntactic dependencies). For instance, the rules of **Fig. 6** say that a LPL between two words can correspond to an *object* dependency in the same direction or a *subject* dependency in the opposite direction.[4]

Linearization rules, which realize the correspondence between syntactic dependencies and linear order, constitute the linearization module. The direction in which the linearization rules are used depends on the input structure given to this module.[5] The dependency tree is at the articulation point between the linearization module and the semantics-syntax interface, while the linear order is at the articulation point between the linearization module and the phonological module. In other words, the linearization module will be called by one of these two other modules: the semantics-syntax interface in synthesis, which gives it a dependency structure as input, or the phonological module in analysis, which gives it a linear order as input.

To ensure that a module has processed the whole input structure, and for modules to call one another, we use a unique device: the polarization of objects, which we will now present.

3.2 Polarized Unification Grammar

Polarized Unification Grammar (henceforth, PUG) is a fomalism inspired by TAG, where rules are modelled by elementary structures that combine to produce the final structure of an utterance. Unlike TAG though, PUG handles any kind of graph, and not only trees. PUG considers four kinds of entities: objects, functions, atomic values, and polarities. The *objects* handled by the linearization module are nodes representing words, and edges representing syntactic dependencies or LPLs.[6] Syntactic dependencies and LPLs are binary edges, which require two nodes as their source and target. Edges are bound to their source and target nodes through *structural functions*, which take as their argument an object and return as their value another object. Two other kinds of functions are considered in PUG, besides structural functions: *labelling functions*, that associate an object with an *atomic value*, and *polarizing functions*, that link an object to a *polarity*. **Fig. 7** shows a fragment of the dependency tree of (2) and its formalization when the structural and labelling functions are made explicit. Note that the visual representation of edges as arrows is merely a convenient way to indicate that this object has a source and a target. Despite this visual metaphor, edges really are objects just like nodes, as **Fig. 7** makes explicit.



≡

$$\begin{bmatrix} \text{label}(1) = \textit{Mary} \\ \text{label}(2) = \textit{subj} \\ \text{label}(3) = \textit{loves} \\ \text{source}(2) = 1 \\ \text{target}(2) = 3 \end{bmatrix}$$

Fig. 7. Making objects and functions explicit

Polarities indicate whether an object must be consumed by a given module. We consider a lot of polarizing functions for our different modules, but we can work with only the same two values for any polarizing function: black and white. A black object, i.e., an object with a black polarity for a given function, represents a resource, while a white object represents a requirement. An object can receive several polarities through as many polarizing functions, generally associated with different modules of the model. When an object has been handled by a given module, it is generally black for the polarizing function pertaining to this module, but it will be white for the polarizing function pertaining to the next module we want to trigger. In other words, the black polarity is *neutral*, which means that a black object does no longer need to be handled, while the white polarity is non neutral, requiring to be saturated by a black polarity. A structure is said to be *saturated* for a given polarizing

[4] The fact that the subject relation has a finite verb as governor could be indicated in this rule, but this constraint is already verified by the semantics-syntax interface (see Section 3.4) and can also be verified by the syntactic well-formedness grammar (see [23]).

[5] We are presenting our grammar in a *transductive* perspective, where an input structure is given to the grammar and the corresponding structure is produced. As we will see in Section 3.6, there are two other ways of considering a correspondence grammar: the *equative* mode, where two structures are given to the grammar and the grammar verifies whether they correspond to each other; and the *generative* mode, where no structure is given to the grammar, which produces couples of structures corresponding to each other.

[6] Previous presentations of PUG distinguished the nodes at different levels of representation. It would be possible here too and even necessary if we wanted to take into account that in some cases, such as amalgams (*ain't* = *am* + *not*), some nodes of a given level can be merged at another level. Here, we consider that the nodes are the same at the different levels of representation and we focus on reordering the graph into a string.

function if the value of this function is neutral for every object of the structure. At the end of the process, the derived structure must be saturated for every polarizing function considered.

An instance of PUG is a finite set of *elementary structures*, with a subset of initial structures. An elementary structure comprises a finite number of objects linked by structural functions (which define the structure proper) and associated to a finite number of labels and polarities (by labelling and polarizing functions). Structures combine by the unification of at least one object. When two objects are unified, the value of every function applying to both objects must be unified too. If the values of a function are atomic values, they must be identical, otherwise unification fails. If they are polarities, they combine by a special operation called the *product on polarities*. The white polarity is the identity for the product (white · white = white, white · black = black), while the product of two black polarities fails (black · black = $\perp$), which means that two black objects cannot be unified. A derivation consists in neutralizing the derived structure. The process starts with any elementary structure, and at each step a new elementary structure is combined with the structure resulting from the previous step. This process can only stop when all the objects are black. Special structures, marked as *initial structures* must be used exactly once.[7]

For the linearization process, we will consider three main grammar modules: G_synt is a grammar verifying that the syntactic structure is a dependency tree, G_string is a grammar verifying that the linear order is a string and G_lin is the linearization module proper. The two first grammars will be presented now.

3.3 Trees and strings in PUG

It is very easy to write a grammar that builds only trees in PUG, i.e., to force the derived graphs to be trees. Such a grammar has only two rules: an initial rule introducing a black node, and another rule introducing a black dependency and a black node as its target.[8] This PUG is represented in **Fig. 8**. The polarity of objects is represented by a triangle next to it, white or black according to its value. Rules are placed in square boxes, with initial rules in double boxes.

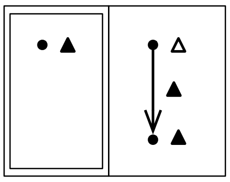


Fig. 8. G_tree, the tree grammar

The initial rule, which must be used exactly once by definition, introduces the only node that will not be governed, i.e., the root of the tree. The connectedness of the structure is ensured by the process itself, which requires that each time a rule is used, at least one object is unified with the structure obtained at the previous step. The second rule ensures that each node except the root has exactly one governor.

G_tree can be used as a generative grammar, generating all possible trees. But in our case, G_tree will be used as a well-formedness grammar, verifying that a given structure is a tree. It can be applied for instance on the syntactic structure of (2) (**Fig. 5**) to verify that it is a tree. To do so, the whole structure will be polarized in white with the polarizing function of the grammar we want to call (cf. the input structure of **Fig. 9**). After the application of G_tree, we obtain a structure entirely black (output structure of **Fig. 9**), which means that the structure has been validated as a well-formed tree.

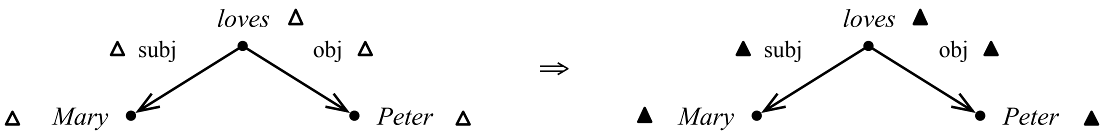


Fig. 9. Input and output structures for G_tree

[7] Such a structure can be compared to the initial non-terminal symbol of a CFG. But contrarily to a CFG, we do not impose the derivation process to start with an initial structure. Moreover, we accept to have several initial structures, which do not increase the generative capacity of the formalism but allows us to write more elegant grammars.

[8] As our objects are not typed, this grammar does not prevent an edge from becoming the vertex of another edge (see [42] for such structures, called polygraphs). This can be solved by adding a labelling function to type objects.

Applying G_tree to our input structure does not only allow us to verify that it is a dependency tree, but also enables us to read the whole input structure and to call other grammars that will apply to it. In our case, we want to articulate G_tree with several grammars, in particular G_lin and G_string. For this, we enrich G_synt to introduce white polarities from these grammars. Let us represent the polarities of G_lin by diamonds, and the ones of G_string by squares.[9] The enriched version of G_tree, which we will call G_synt, will force G_lin to apply on every syntactic dependency and G_string to apply on every nodes.[10] **Fig. 10**a shows G_synt and **Fig. 10**b shows the output structure when G_synt is applied to the syntactic structure of (2).

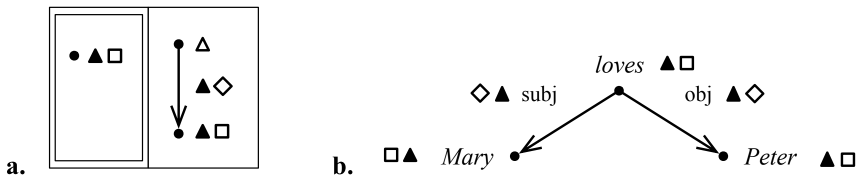


Fig. 10. a. G_synt, a grammar that verifies that every syntactic structure is a tree and calls G_lin and G_string
b. The output for the syntactic structure of (2)

It is a little more difficult to write a grammar that builds only strings. The simplest way is to use two tree grammars, and to verify that the structure is a tree if we read it in both directions. To couple two grammars and force them to apply on the same structure, we just need to add, on each object of each elementary structures of each grammar, a white polarity of the other grammar. These white polarities call the other grammar and ensure that every object has been handled by both grammars. **Fig. 11** shows two tree grammars coupled together. In order to distinguish the polarity pertaining to each grammar, we use an upward triangle for the polarity pertaining to the first tree grammar and a downward triangle for the polarity pertaining to the second grammar.

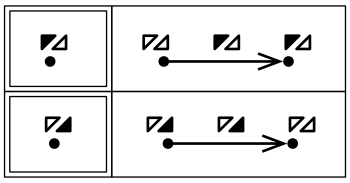


Fig. 11. G_string (first view: two tree grammars coupled together)

After being coupled, the two tree grammars of **Fig. 11** form a single grammar. The rules of the coupled grammar can be used in whatever order, despite the fact that they originally come from two different grammars. This property illustrates one of the major advantages of polarities, which allows to couple different modules in a same model without introducing procedural constraints on the order in which the different modules must be triggered.

It is possible to simplify the previous coupled grammar, because both rules involving an edge must apply on each edge and can be pre-combined. The resulting grammar is given in **Fig. 12**.

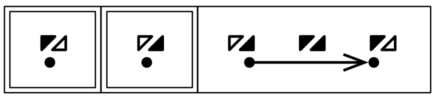


Fig. 12. G_string (second view: pre-combined edge rules)

It is also possible to merge the two polarizing functions. The resulting polarizing function has four possible values: (black, black), (black, white), (white, black), and (white, white). The polarities (black, white) and (white, black) are two opposite polarities, which, combined together, give (black, black), the neutral polarity, indicating that an object is saturated. In previous publications, these polarities have been called *positive* and *negative* [31]

[9] Saying that "the polarities of G_lin are represented by diamonds" is a convenient way to express two things: 1) we consider a new polarizing function, and we use diamonds as a visual convention to represent values of this function; 2) this polarizing function is the one of G_lin because it is mainly rules of G_lin that attribute black polarities as values to this function. But of course this polarizing function is used in rules of all grammars that are articulated with G_lin. For these grammars, these polarities are articulation polarities [23].
[10] We could have introduced less (or more) white polarities. Adding diamond polarities on edges to call G_lin is sufficient. Adding white square polarities on nodes allows us to relax the constraints on possible procedures to trigger our different grammars. With its white square polarities on nodes, G_synt calls G_string, which allows us to trigger G_string before G_lin.

[27] [28]. **Fig. 13** shows G_string after merging the polarizing function. Positive and negative polarities are represented by + and – signs in squares, and (black, black) is represented by a black square.

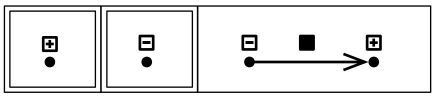


Fig. 13. G_string (third view: polarizing functions merged)

Finally, G_string will be articulated with G_lin and G_synt to be applied to the output of these grammars, so that it triggers them and it is triggered by them, so that the output of one module becomes the input of another. **Fig. 14** presents the resulting grammar, where triangle and diamond polarities pertaining to, respectively, G_lin and G_synt have been added.

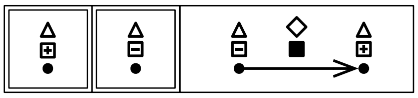


Fig. 14. G_string (fourth view: articulated with G_lin and G_synt)

Note that as soon as the polarities have been added, it is no longer necessary to add the label "<" to distinguish LPLs from syntactic dependencies. The polarities suffice: an edge with a triangle polarity is an edge that must be handled by G_synt, that is, a syntactic dependency, while an edge with a square polarity is an edge that must be handled by G_string, that is, an LPL.[11]

3.4 Sub-categorization rules and semantics-syntax interface

PUG was initially introduced for writing DGs producing unordered dependency trees [26] [29] [32]. **Fig. 15** shows G_sem-synt, a grammar that generates the dependency tree of (1), on semantic grounds. The grammar here is simplified to show only syntactic edges, and morphosyntax (i.e., inflection) is not considered. See [23] or [26] for a more detailed grammar.

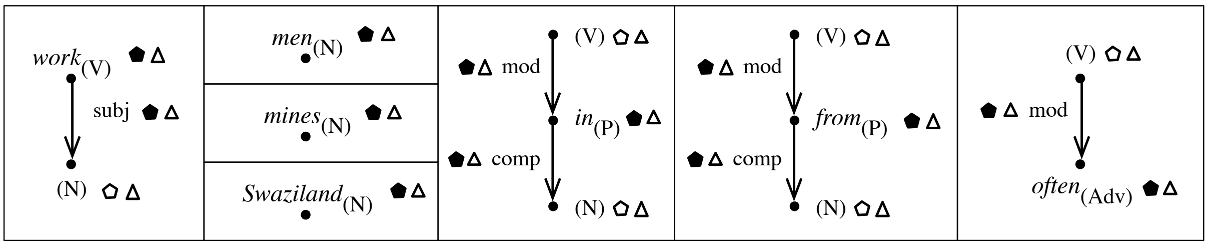


Fig. 15. G_sem-synt, a dependency grammar for (1)

The rule introducing *work* specifies that it is an intransitive verb with only one semantic argument, its subject. The polarities for G_sem-synt are represented by pentagons. White syntactic polarities are added to black pentagons in order to call G_tree. Two labelling functions for nodes are considered: one has a word as value, the other its category, written between parentheses here. The rule for *work* introduces three objects. It says that *work* is of category V and has a subject dependent. The node *work* and the *subj* dependency are resources produced by the rule (they have black pentagons), while the dependent node of category N is a requirement. This need will be filled by the unification with the elementary structure for *men*, which is a black node of category N. The rule for the preposition *in* says that *in* is a preposition that needs a V as governor and an N as dependent. In other words, the elementary structure for *in* will adjoin on *works*. Prepositions *in* and *from* are binary predicates. The adverb *often* is a unary predicate modifying its only argument. Nouns in this sentence have no arguments.

[11] To some extent, it is not even necessary to separate syntactic edges and LPLs. A same edge can be both and bear simultaneously a triangle and a square polarity, exactly as a node can be both a syntactic node and a node of the linear order. It is viewed as a syntactic object when the triangle polarity is handled and as a morphological object when the square polarity is handled. In this paper, edges are moved from node to node and a syntactic dependency can have a different source and target than its corresponding LPL. But we can also understand the process as a move of nodes from edges to edges. In some sense, every edge we consider has two faces: a semantic one (represented here by the syntactic dependency) and a morphological one (represented by an LPL). They describe constructions, that is, linguistic signs with a signified (the semantic face) and a signifier (the morphological face).

It is important to note that G_sem-synt does not force the derived structure to be a tree. This condition is often verified by a hidden procedural device (as in [29] or [32]). In PUG, this is made explicit by forcing the tree grammar G_tree to apply on the syntactic part of the structure.

3.5 Governor-dependent linearization rules

Sketches of the two rules needed to linearize sentence (2) were introduced in **Fig. 6**. To include these rules in PUG we must decide which objects in these rules are resources and which are requirements. Let us call G_lin the linearization grammar that includes these rules. It is a correspondence grammar, like G_sem-synt (Section 3.4), as it puts in correspondence two structures, the syntactic tree and the linear order. Hence, we propose that G_lin produces both structures. Moreover, it calls G_synt to verify that the syntactic part of the structure is a tree, and G_string to verify that the corresponding order is a string. We obtain the rules in **Fig. 16**, where the values of the polarizing function pertaining to G_lin are represented by diamonds. Syntactic dependencies receive a black diamond and a white triangle, to trigger G_synt, while LPLs receive a black diamond and a white square, to trigger G_string. We consider that nodes are not handled by G_lin and are just not polarized.[12]

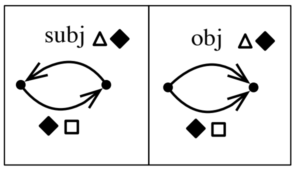


Fig. 16. Governor-dependent linearization rules of G_lin

3.6 Architecture of the model and procedure

The three grammars we have introduced, G_synt, G_string and G_lin, call one another. This illustrates the fact that PUG can formalize complex architectures and is not restricted to pipeline architectures where modules form a chain. PUG has the advantage of not forcing any procedure.[13]

In fact, our three grammars are now gathered in only one grammar, which we call G_synt × G_lin × G_string. The three grammars are still visible as three modules of our combined grammar, but we do not need to separate them. Elementary trees of the three grammars can be triggered in whatever order we want. In particular, the combined grammar is reversible and can be used for both synthesis and analysis. In synthesis, G_synt is triggered first to verify that the input structure is a dependency tree, then G_lin associates a linear order to it, and finally G_string verifies that the output is a string. In analysis, the reverse occurs: G_string is triggered first, calling G_lin for the correspondence, and G_synt verifies the well-formedness of the output. It is also possible to trigger G_synt and G_string first, and only then verify that the generated tree and string are compatible via G_lin. It is as well possible to mix the grammars and to alternate rules of the different modules. For instance, we can use the grammar for incremental parsing by building the syntactic tree edge by edge as we are consuming the string. Whatever order we choose, the polarization ensures that, if we obtain a neutral structure at the end, we will have a tree and a string corresponding to each other by our linearization rules and that the three modules of the grammar have been applied.

There are three main ways to use a correspondence grammar such as G_lin or G_synt × G_lin × G_string [33]. First, the grammar can be used as a purely *generative* process, building couples of trees and strings corresponding to one another, from scratch. In our case, the grammar will be used in a *transductive* way, transforming a tree into a string, or a string into a tree. In these cases, the process starts with one of the two structures. To trigger the grammar, this input structure is polarized in white: with the white polarity of G_string if we suppose that it is a string, and with the white polarity of G_synt if we suppose that it is a dependency tree. The "transduction" of the input structure into another one will be automatically achieved by the need to neutralize it. There is a third way to use the grammar: the *equative* way. In this case, a couple of structures is considered and the grammar will verify that they are a tree and a string and that they correspond to each other by trying to neutralize both of them. [14] said that a Meaning-Text model "is by no means a generative or, for that

[12] In [28], such objects received a neutral grey polarity that was the identity for the product on polarities, which is equivalent to having no polarity.
[13] It must nevertheless be noted that the space of possible procedures is controlled by the articulation polarities we add in the rules of each grammars. In this presentation we chose to leave open a maximum number of possibilities: each time an object in a rule can be handled by another grammar, we add an articulation polarity to call this grammar. But it is possible to restrict the use of articulation polarities in order to have a pipeline architecture.

matter, transformational system: it is a purely EQUATIVE (or translative) device" (p. 45). But in fact, correspondence grammars, like modules of MTT, can be used in all three modes—equative, transductive, and generative. In particular, our correspondence grammars are reversible and can be used for synthesis as well as analysis.

4 Linearization module

The linearization grammar presented so far is not sufficient to order all dependency trees. Our first linearization module (Section 3.5) contained only governor-dependent linearization rules, i.e., rules specifying the position of a node in relation to its governor. A second set of rules is needed to position co-dependents in relation to each other (Section 4.1). A third set of rules propagates LPLs to nodes that are not in a so close relation in the dependency tree (Section 4.2). Non-projective orders will be studied in Section 5.

4.1 Co-dependent linearization rules

Let us take an example:

(3) *Mary often works.*

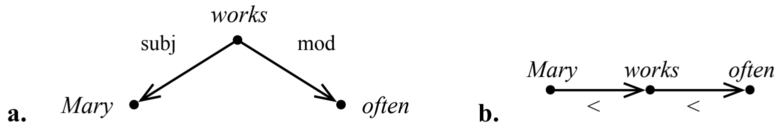


Fig. 17. a. Dependency tree **b.** linear order of (3)

Fig. 17 shows the dependency tree and linear order of (3). The subject (*Mary*) and the modifier (*often*) are both on the left of their governor (*works*), but the subject must be before the modifier. The subject dependency between *Mary* and *works* now corresponds to a LPL between *Mary* and *often*. This means that we need a linearization rule involving these three nodes—*Mary*, *works*, and *often*—and both syntactic dependencies—*subj* and *mod*. This is the first rule in **Fig. 18**. It associates the *subj* dependency with an LPL between two co-dependents. Both edges are built by this rule, and thus have a black diamond polarity, while the *mod* dependency has a white diamond polarity because it will be associated with an LPL by another rule. This latter rule is the second in **Fig. 18**. It has category constraints and only applies to an adverb (Adv) modifying a verb (V).

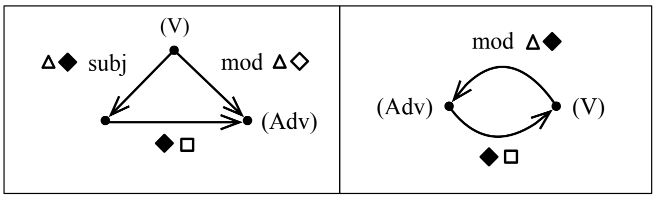


Fig. 18. The *subj-mod* co-dependent and the *mod* governor-dependent linearization rules of G_lin

Fig. 19 shows the application of these two rules for (3). The output structure contains a dependency tree and a linear order. Dependencies have a white triangle polarity calling G_synt and LPLs have a white triangle polarity calling G_string. The structure is presented twice: once with a dependency tree-based layout (**Fig. 19**a), the other with a linear order-based layout (**Fig. 19**b). Let us emphasize that the two schemata are just different views of the exact same graph.

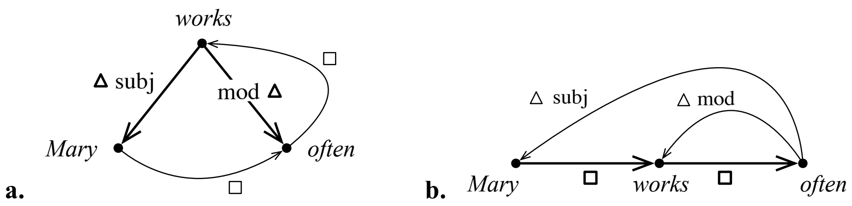


Fig. 19. Syntactic dependencies and LPLs of (3) after application of G_lin

The *subj* governor-dependent linearization rule of **Fig. 18**, placing the subject before the verb, is still needed because the adverb can be absent. If this rule is applied to the syntactic tree of (3) instead of the *subj-mod* co-dependent linearization rule, we will obtain a structure that is not a string and will be rejected by G_string.

The grammar presented here only handles the linearization of what we will call direct projections. The *direct projection* of a node *x* in a dependency tree is the node itself and its direct dependents. For instance, the direct projection of *works* in (1) is *men often work in*. We will see in Section 4 how to propagate the LPLs between the words of the direct projections to the whole set of nodes. Before that, we will see how to homogenize the rules used to order direct projections.

4.2 Propagation and projectivity

After applying governor-dependent and co-dependent linearization rules on the dependency tree of (1), we obtain the structure in **Fig. 20**. In this figure, we only kept the articulation polarities: triangle polarities for syntactic dependencies and square polarities for LPLs (the diamond polarities are all black). As before, this structure is presented twice, in a dependency tree-based layout (**Fig. 20**a) and in a linear order-based layout (**Fig. 20**b).

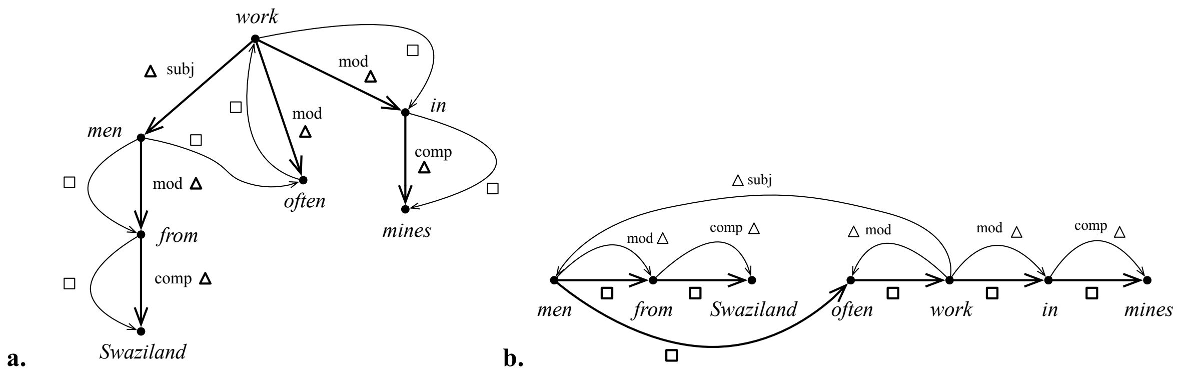


Fig. 20. Syntactic dependencies and LPLs of (1) after application of G_lin

As we can see, we cannot obtain a string with the application of only governor-dependent and co-dependent linearization rules. We know that *men* is before its dependent *from* and its co-dependent *often*, but we do not know the relative order of *from* and *often*, and this order cannot be computed by our current G_lin, because *from* and *often* are in an "aunt-niece" relation in the dependency tree, and G_lin covers only governor-dependent (i.e., mother-daughter) and co-dependent (i.e., sister) relations.

To obtain the complete linearization of the dependency tree, we must use a property of the correspondence between a dependency tree and a linear order: projectivity. There are several equivalent definitions of projectivity. The definition we use, stated by [34], who coined the term *projectivity*, is based on the notion of projection. The *maximal projection* of a node is the set formed by the node itself and all the nodes it dominates (directly or indirectly) in the dependency tree. A linearly ordered tree is *projective* if and only if the maximal projection of every node of the dependency tree is continuous in the linear order.

A consequence of projectivity is that, if a node *x* is before a node *y* that is not in the projection of *x*, then the whole projection of *x* is before *y* and, in particular, every dependent of *x* is before *y*. This property (and the symmetric property where *x* is after *y*) can be translated into a rule (**Fig. 21**). This rule says that any LPL from a node *x* to a node *y* can be replaced by an LPL from any dependent of *x* to *y*. The LPL between *x* and *y* receives a black square polarity, which means that it can no longer be covered by G_string, while the new LPL that replaces it receives a white square polarity and is active for G_string. In analysis, the rule is read in the reverse order: any LPL from a node *z* to a node *y* can be replaced by an LPL from the governor of *z* to *y*. The former LPL receives a black diamond polarity and is no longer active for G_lin, while the new LPL that replaces it receives a white diamond polarity and is active for G_lin. The white triangle polarity between the sources of the two LPLs of rules of **Fig. 21** forces these two nodes to be linked by a syntactic dependency (which will be neutralized by G_synt).

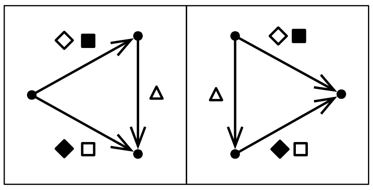


Fig. 21. Propagation rules of G_lin

Propagation rules propagate LPLs downwards in synthesis and upwards in analysis. **Fig. 22** shows their application in synthesis for a fragment of the structure in **Fig. 20**. The first rule of **Fig. 21** is applied twice to propagate the LPL between *men* and *often* to *from* and then to *Swaziland*. The output is a string and will be saturated (i.e., accepted) by G_string.

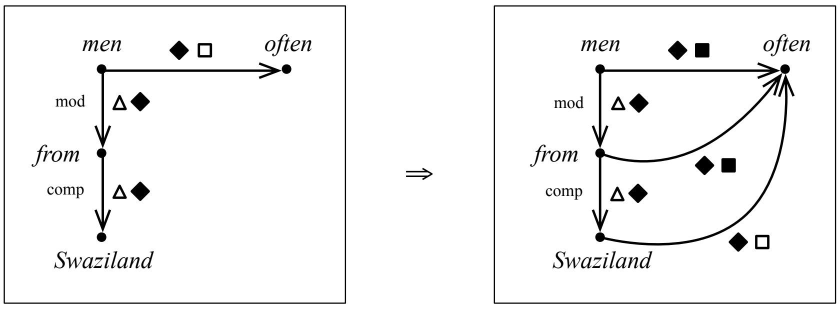


Fig. 22. Application of propagation rules on the structure of (1)

Let us make a remark about the set of LPLs built by our grammar. Order is a transitive relation. From a computational point of view (and certainly also from a cognitive point of view), it is not necessary to grasp all the LPLs that a linear order implies. As we see here, we need mainly to consider immediate LPLs, i.e., LPLs between two successive words, and some other LPLs that result from the propagation of immediate LPLs. Thus, propagation is a kind of transitivation of linear order, but highly constrained by syntactic structure.

The previous remark is illustrated by **Fig. 23**, which shows the application of the linearization module for the analysis of (1). The input structure is the string of words, which contains only immediate LPLs (cf. **Fig. 1**). The application of G_string introduces white diamond polarities on immediate LPLs, which call G_lin (**Fig. 23**a). After applying all the rules of G_lin that can neutralize a white diamond polarity now, we obtain the structure in **Fig. 23**b. Only one LPL has not been neutralized, the LPL from *Swaziland* to *often*. Neither governor-dependent nor co-dependent linearization rules can be triggered on this LPL. But this LPL can be propagated, producing the configuration in **Fig. 23**c, on which the *subj-mod* co-dependent linearization rule can now apply, producing the output structure in **Fig. 23**d. This structure contains the whole dependency tree.

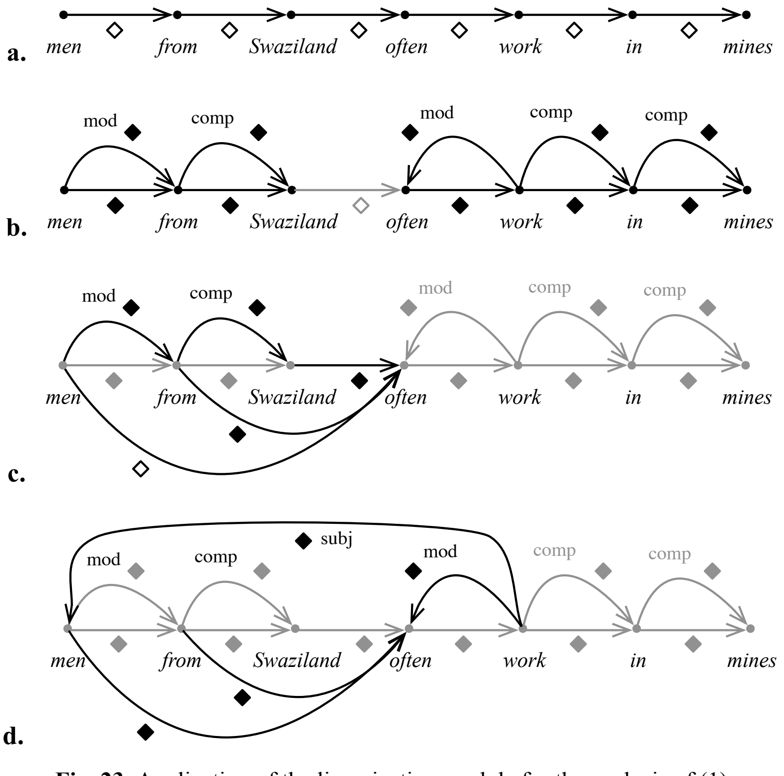


Fig. 23. Application of the linearization module for the analysis of (1)

Propagation rules can only produce a projective structure and they suffice to linearize any output of governor-dependent and co-dependent rules. They would apply to co-dependents if we did not take necessary precautions. In fact, any co-dependent linearization rule is the combination of a governor-dependent linearization rule and a propagation rule, as shown in **Fig. 24**. When the relative order of two co-dependents is free, it can be realized by propagation rules, but when it is not free, then propagation rules must be blocked. This could be done by adding a special feature to the LPL introduced by a governor-dependent rule, as well as the LPL consumed by a propagation rule. In other words, the two LPLs that unify in **Fig. 24** would need to have a different value for this special feature (not represented here) if we wanted to block this unification.

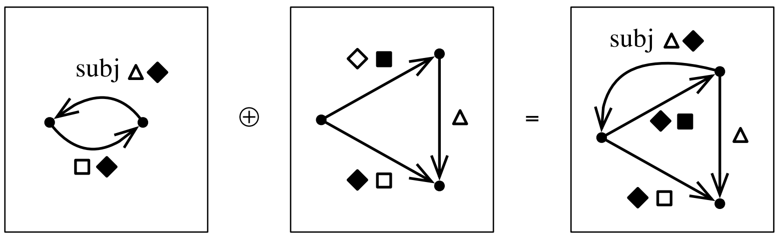


Fig. 24. Co-dependent linearization rules as combinations of governor-dependent linearization and propagation rules

5 Emancipation

Non-projective correspondence between a dependency tree and linear order is common in natural language. In English, it is illustrated by so-called extraction phenomena, like the anteposition of a complement governed by a subordinate verb outside the subordinate clause (cf. sentence (4) and its linearized dependency tree in **Fig. 25**).

(4) *To Mary Peter thinks we should not speak again.*

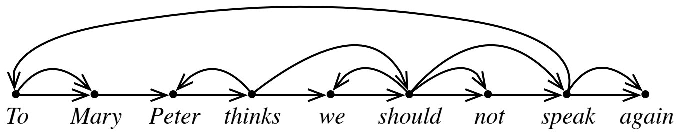


Fig. 25. Non-projective linearized dependency tree of (4)

Such sentences are quite frequent in German, because the first element of a sentence can easily be the dependent of a subordinate verb, as illustrated in (5) (see **Fig. 26**a,b).

(5) *Das Buch*	*hat*	*diesem Mann*	*niemand*	*zu lesen*	*versprochen*
the$_{ACC}$ book	has	this$_{DAT}$ man	nobody$_{NOM}$	to read	promised

‘Nobody promised this man to read the book’

In almost any formalism, non-projective structures are solved by the raising of problematic elements in the syntactic structure. This is achieved by similar devices in all theories: movements in Generative Grammar (Move α [35]), non local features in HPSG (the slash feature [36]), functional uncertainty in LFG [37], etc. In DG, the problem can be solved by raising problematic elements in the syntactic dependency tree [10] [38] [39]. The idea underlying raising in all of these frameworks is that non-projectivity occurs when an element is not positioned in relation with its direct governor, but one of its indirect governors. For instance, in (4), *to Mary* is not positioned in relation to its governor *speak*, but to *thinks*, the main verb. In (5), *das Buch* ‘the book’ is not positioned in relation to its governor *zu lesen* ‘to read’, but to *hat* ‘has’, the main verb (**Fig. 26**).

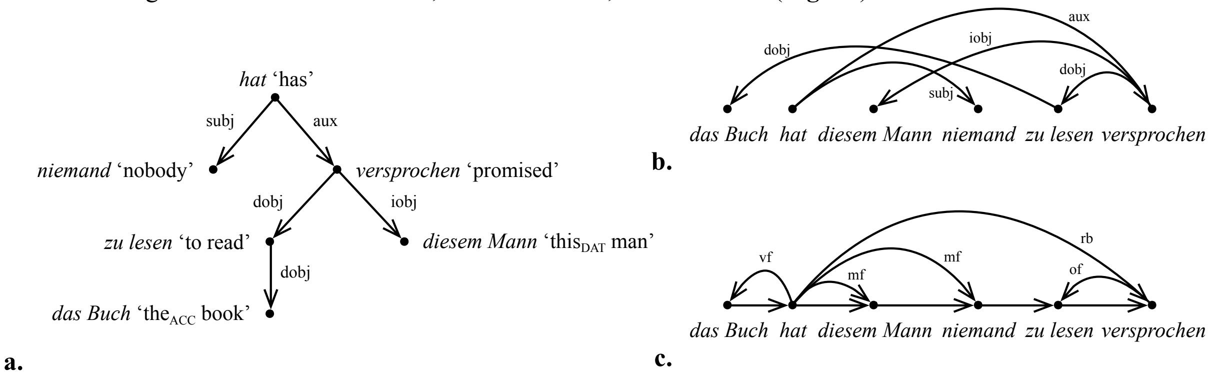

hat ‘has’
subj
aux
niemand ‘nobody’
versprochen ‘promised’
dobj
iobj
zu lesen ‘to read’
diesem Mann ‘this$_{DAT}$ man’
dobj
das Buch ‘the$_{ACC}$ book’
a.
aux
iobj
dobj
dobj
subj
das Buch hat diesem Mann niemand zu lesen versprochen
b.
rb
vf
mf
mf
of
das Buch hat diesem Mann niemand zu lesen versprochen
c.

Fig. 26. a. The dependency tree of (5) **b.** Its non-projective linearization **c.** The corresponding projective topological tree

There are different ways to control raising. In Generative Grammar, the movement is constrained by syntactic categories: some syntactic constituents are “islands” and it is not possible to cross their boundaries (see [40] for a first description of island constraints). In LFG or traditional DG, the constraints are expressed in terms of syntactic functions: the chain of syntactic dependencies between the raised element and the ancestor that “hosts” it in the linear order is constrained by the nature of their syntactic functions [14] [15].

In this paper we adopt a third solution: the topological model. This model has been developed during the 19th century for the description of word order in German and has been formalized in HPSG [41] and DG [11] [12]. It elegantly models the fact that German is a V2 language (the main verb of a declarative sentence always occupies the second position), with a verb cluster at the end of the sentence, possibly followed by some extraposed heavy constituents. This is modeled by decomposing German sentences into five fields (Vorfeld, left bracket, Mittelfeld, right bracket, and Nachfeld = vf, lb, mf, rb, and nf), with the following conditions: the main verb goes in the left bracket, the other verbs in the right bracket, one constituent in the Vorfeld, the others in the Mittelfeld, and some heavy constituents in the Nachfeld. In the verb cluster, each verb is placed to the left of its governor, in a field called the Oberfeld (of). Noun phrases cannot be placed in the verb cluster and are placed between the main verb and the verb cluster. If they depend on a verb in the verb cluster, they must emancipate and go in one of the major fields (Vorfeld, Mittelfeld, or Nachfeld).

In (5), the main verb is the finite auxiliary *hat* 'has', which must be in second position, in the left bracket. Its verbal dependent, the participle *versprochen* 'promised', is placed in the right bracket, where it forms a verb cluster accommodating its verbal dependent *zu lessen* 'to read'. The noun phrases of these two verbs cannot be placed in the verb cluster and will be emancipated to be placed in fields opened by the main verb: *das Buch* 'the book' goes in the Vorfeld, while *diesem Mann* 'to this man' is placed in the Mittelfeld, where it joins the subject *niemand* 'nobody' and can be ordered freely in relation to it.

The topological structure can be represented by a constituent structure as in [12] or by a dependency tree as in [11]. This second representation is adopted here (**Fig. 26**c). The topological tree is added as an intermediate structure between the syntactic tree and linear order. This new structure receives its own polarity, represented by a downward triangle (▽), to be contrasted with the upward triangle (△) of syntactic trees. The diamond (◇) is still used for the linearization module, which now ensures the correspondence between the topological tree and linear order, but uses the same rules as the ones described in previous sections, because the topological tree corresponds to the linear order projectively. A new polarity, represented by a circle (○), is introduced for the syntax-topology interface, which ensures the correspondence between the syntactic tree and the topological tree. **Fig. 27** summarizes the architecture of the grammar with all the polarities: those pertaining to the well-formedness grammars (△ , ▽ , □) and those pertaining to the interfaces (⬠ , ○ , ◇).[14]

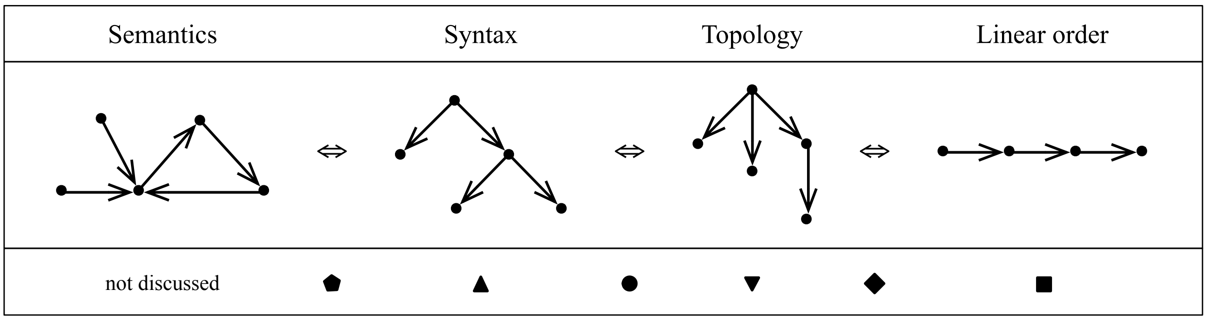


Fig. 27. The modules and their polarities

The rules of the syntax-topology interface are very similar to the rules of the linearization module: the majority are correspondence rules associating a syntactic edge to a topological edge. The other rules resolve mismatches between the two structures. For the linearization module, they are propagation rules. For the syntax-topology interface, they are emancipation rules, which are very similar to propagation rules.[15] **Fig. 28** shows examples of different rules used in the syntax-linear order correspondence: the first two transform a syntactic *aux* edge into a topological *rb* edge (allowing the dependent of the auxiliary to go in the right bracket) and a syntactic *dobj* into a topological *vf* (allowing a direct object to go in the Vorfeld); the next rule is an emancipation rule lifting a *vf* and allowing a node placed in the Vorfeld to emancipate from the right bracket (note that the lower *vf* is only visible for the syntax-topology interface, while the upper *vf* is only visible for the topological well-formedness module, as shown by their polarities); the following rules are topological well-formedness rules verifying that the topological structure is a tree and the Vorfeld is in the main domain; the last rules are linearization rules placing the Vorfeld to the left and the right bracket after the Mittelfeld, as well as a propagation rule.

[14] We have not developed the semantics-syntax interface nor introduced a convention for the semantic polarity in this paper.

[15] The semantics-syntax interface also contains mismatch rules for phenomena such as raising (*Peter seems to sleep*), auxiliaries (*I will read*), *tough*-movement (*a paper hard to read*), or extraction (*a paper I would like to read*). See [32] [26] [25].

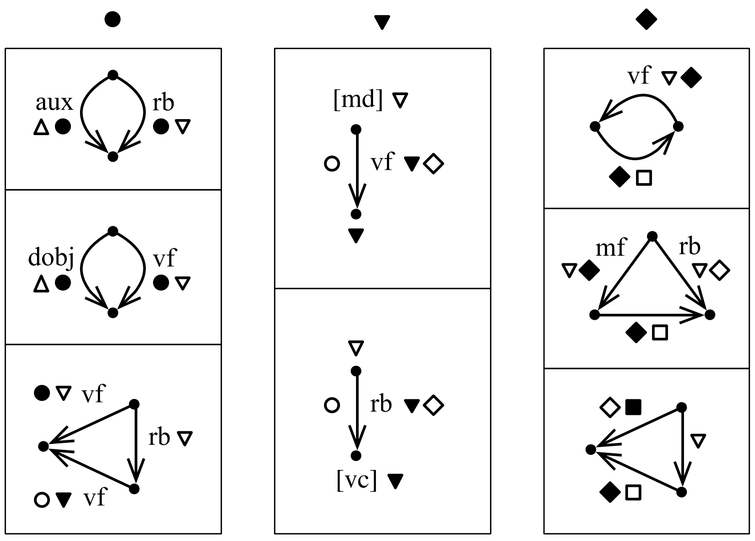


Fig. 28. Rules of syntax-topology-linear order correspondence

The application of syntax-topology correspondence rules produces a copy of the syntactic tree where the node and edge labels have changed. The application of the emancipation rules lifts some edges (**Fig. 29**). The emancipation could have been done directly in the syntactic tree, but the advantage of relabelling the tree in the syntax-topology is to have to different grammar for the well-formedness of the syntactic tree and the topological tree.

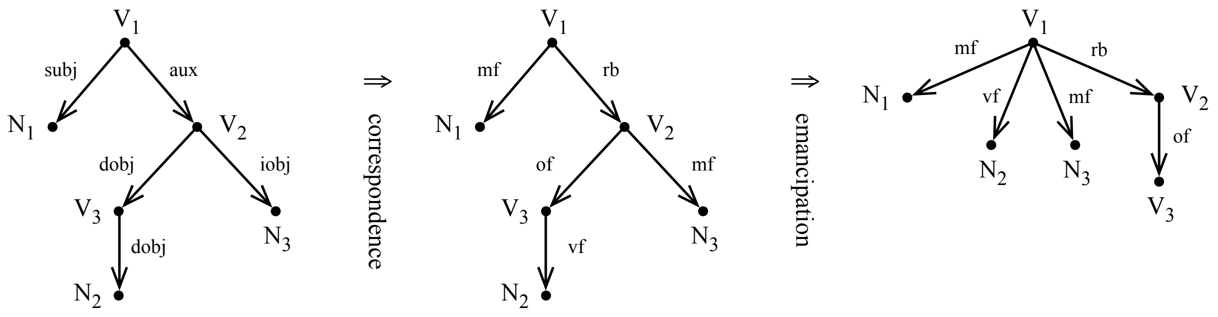


Fig. 29. The syntax-topology interface applied on the trees of (5)

A more complete topological model for German can be found in [12].

6 Conclusion

This paper pursued several goals: showing that linearization can be seen as a graph rewriting process, showing that each LPL corresponds to a syntactic dependency and that the path between the two can be traced, and formalizing the linearization module in a modular, declarative formalism, allowing to combine it with other modules of a linguistic model.

The paper gives a large overview of Polarized Unification Grammar. The formalism is well adapted to writing grammars that generate various structures, such as strings, trees, and graphs, and correspondence grammars between such structures. Polarization allows us to split the model into small modules articulated with one another, and to maintain a distributed architecture where every module calls the other modules to handle the same structures. PUG also allows us to deal with node expansions, which has not been developed here.

From a theoretical point of view, this paper proposes a linguistic model with several levels of representation, including a syntactic level where the structure is an unordered dependency tree. The choice of dependency trees for the representation of syntax is supported by two arguments. First, it elegantly interfaces with both semantics and linear order, including non-projective orders. Second, there is a one-to-one correspondence between the syntactic dependencies and the linear precedence links between couples of successive words, and this correspondence can be exploited in various ways, in particular to predict prosodic breaks.

From a formal point of view, we have a quite complex architecture, but the articulation between the different modules is ensured by a single mechanism (polarization), which allows us to control rule combination. This is done without imposing any order on the combination of rules, thus preserving a distributed architecture. The typology of the rules we introduce is also quite simple. We have well-formedness rules verifying that the structure is well-formed (for instance that it is a tree or a string), correspondence rules transforming a structure in another one, and propagation/emancipation rules in case of mismatches.

We did not address the implementation of PUG. In fact, we tried to underspecify procedure as much as possible. Our grammar allows various processing chains, and it was not our purpose to decide which procedure is better for synthesis or parsing. On the order hand, our grammar is very precise on which word order specification must be computed to linearize a dependency tree. We showed that we only need to consider LPLs between successive words, between nodes linked by a syntactic dependency, and between all couples of words that propagation rules may go through. Although no procedure is given, the set of objects that any procedure would have to handle and the set of elementary operations that must be triggered is clearly delimited. Moreover, the word order we produce is explicit (it is a linear graph on words).

References

1. Gazdar, G.: Unbounded dependencies and coordinate structure. Linguistics Inquiry 12(1), 155–184 (1981)
2. Gazdar, G., Klein, E., Pullum, G., Sag, I.: Generalized Phrase Structure Grammar. Harvard University Press, Cambridge (1985)
3. Candito, M.-H.: A principle-based hierarchical representation of LTAG. In : Proceedings of Coling, Copenhagen (1996)
4. Kaplan, R., Bresnan, J.: Lexical-Functional Grammar: A Formal System for Grammatical Representation. In Bresnan, J., ed. : The Mental Representation of Grammatical Relations. The MIT Press, Cambridge (1982) 173–281
5. Bresnan, J.: Lexical-Functional Syntax. Blackwell, Malden (2001)
6. Andreas, K.: Linerarization-based German Syntax. Ph.D. thesis, Ohio State University (1995)
7. Richter, F., Sailer, M.: Remarks on Linearization. Reflections on the Treatment of LP-Rules in HPSG in a Typed Feature Logic. Master's dissertation, Eberhard-Karls-Universität, Tübingen (1995)
8. Müller, S., Kasper, W.: HPSG analysis of German. In : Verbmobil: Foundations of Speech-to-Speech Translation. Springer, Berlin (2000) 238–253
9. Tesnière, L.: Éléments de syntaxe structurale. Klincksieck, Paris (1959)
10. Kahane, S., Nasr, A., Rambow, O.: Pseudo-projectivity: a polynomially parsable non-projective dependency grammar. In : Proceedings of Coling-ACL, Montreal, pp.646–652 (1998)
11. Duchier, D., Debusmann, R.: Topological dependency trees: a constraint-based account of linear precedence. In : Proceedings of the 39th Annual Meeting on Association for Computational Linguistics, Toulouse, pp.180–187 (2001)
12. Gerdes, K., Kahane, S.: Word order in German: a formal dependency grammar using a topological hierarchy. In : Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (ACL-01), Toulouse, pp.220–227 (2001)
13. Kuhlmann, M., Nivre, J.: Transition-based techniques for non-projective dependency parsing. Northern European Journal of Language Technology 2(1), 1–19 (2010)
14. Mel'čuk, I.: Dependency syntax: theory and practice. State University of New York Press, Albany (1988)
15. Mel'čuk, I., Pertsov, N.: Surface Syntax of English: A Formal Model Within the Meaning-Text Framework. John Benjamins, Amsterdam (1987)
16. Iordanskaja, L., Mel'čuk, I.: Ordering of Simple Clauses in an English Complex Sentence. Rhema(4), 17–59 (2015)
17. Mel'čuk, I.: Semantics: From Meaning to Text 1. John Benjamins, Amsterdam (2012)
18. Tesnière, L.: Elements of structural syntax. John Benjamins, Amsterdam (2015)
19. Mel'čuk, I.: Actants in Semantics and Syntax I: Actants in Semantics. Linguistics 42(1), 1–66 (2004)
20. Kahane, S.: Dependency and Valency: An International Handbook of Contemporary Research. Walter de Gruyter, Berlin (2003)
21. Copestake, A.: Slacker semantics: why superficiality, dependency and avoidance of commitment can be the right way to go. In : Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics, Athens, pp.1–9 (2009)
22. Kahane, S., Mel'čuk, I.: Synthèse des phrases à extraction en français contemporain (du réseau sémantique à l'arbre syntaxique). Traitement Automatique des Langues 40(2), 25–85 (1999)
23. Kahane, S., Lareau, F.: Meaning-Text Unification Grammar: modularity and polarization. In : Proceedings

of the Second International Conference on Meaning-Text Theory, Moscow, pp.163–173 (2005)

24. Lareau, F.: Vers une grammaire d'unification Sens-Texte du français: le temps verbal dans l'interface sémantique-syntaxe. Ph.D. thesis, Université de Montréal / Université Paris 7 (2008)
25. Lareau, F.: Le temps verbal dans l'interface sémantique-syntaxe du français. In : Proceedings of the Fourth International Conference on Meaning-Text Theory, Barcelona (2009)
26. Kahane, S.: Predicative Adjunction in a Modular Dependency Grammar. In : Proceedings of the 2nd international conference on Dependency Linguistics (DepLing), Prague, pp.137–146 (2013)
27. Kahane, S.: Grammaires d'Unification Polarisées. In : Actes de la 11ème conférence sur le Traitement Automatique des Langues Naturelles, Fès, pp.233–242 (2004)
28. Kahane, S.: Polarized Unification Grammars. In : Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Sydney, pp.137–144 (2006)
29. Nasr, A.: A formalism and a parser for lexicalized dependency grammars. In : 4th International Workshop on Parsing Technologies, Prague, pp.186–195 (1995)
30. Perrier, G.: Interaction grammars. In : Proceedings of the 18th International Conference on Computational Linguistics, Saarbrücken, pp.600–606 (2000)
31. Duchier, D., Thater, S.: Parsing with tree descriptions: a constraint-based approach. In : Proceedings of the Sixth International Workshop on Natural Language Understanding and Logic Programming (NLULP), Las Cruces, NM, pp.17–32 (1999)
32. Kahane, S.: A fully lexicalized grammar for French based on Meaning-Text Theory. In : Proceedings of Cicling: Computational Linguistics and Intelligent Text Processing, Mexico, pp.18–31 (2001)
33. Sylvain, K.: Des grammaires formelles pour définir une correspondance. In : Actes de la 7e conférence annuelle sur le Traitement Automatique des Langues Naturelles (TALN), Lausanne (2000)
34. Lecerf, Y.: Une représentation algébrique de la structure des phrases dans diverses langues natuelles. Comptes Rendus de l'Académie des Sciences de Paris 252, 232–234 (1961)
35. Chomsky, N.: The Minimalist Program. The MIT Press, Cambridge (1995)
36. Pollard, C., Sag, I.: Head-Driven Phrase Structure Grammar. CSLI, Stanford (1994)
37. Kaplan, R., Zaenen, A.: Long-distance dependencies, constituent structure, and functional uncertainty. In Baltin, M., Kroch, A., eds. : Alternative Conceptions of Phrase Structure. Chicago University Press, Chicago (1989) 17–42
38. Bröker, N.: Unordered and non-projective dependency grammars. Traitement Automatique des Langues 41(1), 245–272 (2000)
39. Hudson, R.: Discontinuity. Traitement Automatique des Langues 41(1), 15–56
40. Ross, J.: Constraints on Variables in Syntax. Ph.D. thesis, MIT, Cambridge, MA (1967)
41. Kathol, A.: Linerarization-based German Syntax. Ph.D. thesis, Ohio State University (1995)
42. Kahane, S., Mazziotta, N.: Syntactic polygraphs: A formalism extending both constituency and dependency. In : Proceedings of the 14th Meeting on the Mathematics of Language, Chicago, pp.152–164 (2015)