

Dependency Structures Derived from Minimalist Grammars

Marisa Ferrara Boston
Cornell University
Ithaca, NY
mfb74@cornell.edu

John T. Hale
Cornell University
Ithaca, NY
jthale@cornell.edu

Marco Kuhlmann
Uppsala University
Uppsala, Sweden
marco.kuhlmann@lingfil.uu.se

Abstract

This paper provides an interpretation of Minimalist Grammars (Stabler, 1997; Stabler & Keenan, 2003) in terms of dependency structures. Under this interpretation, merge operations derive projective dependency structures, and movement operations create both non-projective and illnested structures. This provides a new characterization of the generative capacity of Minimalist Grammar, and makes it possible to discuss the linguistic relevance of non-projectivity and illnestedness based on grammars that derive structures with these properties.

1 Introduction

This paper investigates the class of dependency structures that Minimalist Grammars (MGs) (Stabler, 1997; Stabler & Keenan, 2003) derive. MGs stem from the generative linguistic tradition, and Chomsky’s Minimalist Program (1995) in particular. The MG formalism encourages a lexicalist analysis in which hierarchical syntactic structure is built when licensed by word-properties called “features”. MGs facilitate a movement analysis of long-distance dependency. The movement rule, too, is conditioned by lexical features. Unlike unification grammars, but similar to categorial grammar, these features must be cancelled in a particular order specific to each lexical item.

Dependency Grammar (Tesnière, 1959) is a linguistic formalism that determines sentence structure on the basis of word-to-word connections, or dependencies. DG names a family of approaches to syntactic analysis that all share a commitment to word-to-word connections. Kuhlmann (2007) relates dependency graph properties like projectivity and wellnestedness to language-theoretic concerns like generative capacity.

This paper examines these same properties in MG languages. We do so using a new DG interpretation of MG derivations. This tool reveals that syntactic movement as formalized in MGs can derive the sorts of illnested structures attested in Czech comparatives from the Prague Dependency Treebank 2.0 (PDT) (Hajič et al., 2000). Previous research in the field indirectly link MGs and DGs: Michaelis (2001) proves the equivalence of MGs with Linear Context-Free Rewriting Systems (LCFRS) (Vijay-shanker et al., 1987), and the relation of LCFRS to DGs is made explicit in Kuhlmann (2007). This paper provides a more direct connection between the two formalisms. Using this connection, we investigate the linguistic relevance of structural constraints such as non-projectivity and illnestedness based on MGs that induce structures with these properties. In this way, the system proposed is not a new formalism, but a technical tool that we use to gain linguistic insight.

Section 2 describes MGs as they are formalized in Stabler & Keenan (2003), and Section 3 translates MG operations into operations on dependency structures. Sections 4 and 5 discuss the dependency structural constraints of projectivity and nestedness in terms of MGs.

$$\begin{array}{c}
\frac{s :: =f\gamma \quad t \cdot f, \alpha_1, \dots, \alpha_k}{st : \gamma, \alpha_1, \dots, \alpha_k} \text{merge1} \\
\frac{s : =f\gamma, \alpha_1, \dots, \alpha_k \quad t \cdot f, \iota_1, \dots, \iota_l}{ts : \gamma, \alpha_1, \dots, \alpha_k, \iota_1, \dots, \iota_l} \text{merge2} \\
\frac{s \cdot =f\gamma, \alpha_1, \dots, \alpha_k \quad t \cdot f\delta, \iota_1, \dots, \iota_l}{s : \gamma, \alpha_1, \dots, \alpha_k, t : \delta, \iota_1, \dots, \iota_l} \text{merge3} \\
\frac{s : +f\gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \dots, \alpha_k}{ts : \gamma, \alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_k} \text{move1} \\
\frac{s \cdot +f\gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \dots, \alpha_k}{s : \gamma, \alpha_1, \dots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \dots, \alpha_k} \text{move2}
\end{array}$$

Table 1: Merge and Move

2 Minimalist Grammars

This section introduces notation particular to the MG formalism. Following Stabler (1997) and Stabler & Keenan (2003), a Minimalist Grammar G is a five-tuple $(\Sigma, F, Types, Lex, \mathcal{F})$. Σ is the vocabulary of the grammar, which can include empty elements. Figure 6 exemplifies the use of empty “functional” elements that is typical in Chomskyan and Kaynian analyses. There ϵ denotes an empty element that, while syntactically potent, makes no contribution to the derived string. F is a set of *features*, built over a non-empty set of *base features*, which denote the lexical category of the item. If f is a base feature, then $=f$ is a *selection feature*, which selects for complements with base feature f ; a prefixed $+$ or $-$ identifies licensor and licensee features, $+f$ and $-f$ respectively, that license movement. G distinguishes two *types* of structure. The “simple” type, flagged by double colons ($::$), identifies items fresh out of the lexicon. Any involvement in structure-building creates a “derived” item ($:$) flagged with a single colon. This distinction allows the first derivation of syntactic composition to be handled differently from later episodes. A *chain* is a triple $\Sigma^* \times Types \times F^*$, and an *expression* is a non-empty sequence of chains. The set of all expressions is denoted by E . The *lexicon* Lex is a finite subset of chains with type $::$. The set \mathcal{F} is a set of two generating functions, **merge** and **move**. For simplicity, we focus on MGs that do not incorporate head or covert movement. Table 1 presents the functions in inference-rule form, following Stabler & Keenan (2003). In Table 1, the juxtaposition st denotes the concatenation of two strings s and t .

Merge is a structure building operation that creates a new, derived expression from two expressions ($E \times E \rightarrow E$). It is the union of three functions, shown in the upper half of Table 1. Each sub-function applies according to the type and feature of the lexical items to be merged. The three merge operations differ based on the types of chains that are taking part in the operations. If s is simple ($::$), the **merge1** operation applies. If it is derived ($:$), the **merge2** operation applies. We write \cdot when the type does not matter. If t has additional features δ , the **merge3** operation must apply regardless of the type of s . The **move** operation is a structure building operation that creates a new expression from an expression ($E \rightarrow E$). It is the union of two functions, **move1** and **move2**, provided in the lower half of Table 1. As with the merge operations, **move2** only applies when t has additional features δ .

3 MG operations on dependency trees

In this section we introduce a formalism to derive dependency structures from MGs. Throughout the discussion \mathbb{N} denotes the set of non-negative integers.

$$\begin{array}{c}
\frac{(\{\lambda\}, \langle \lambda \rangle) ::= f\gamma \quad (\mathbf{t}, x) \cdot f, \alpha_1, \dots, \alpha_k}{(\{\lambda\} \cup \uparrow_i \mathbf{t}, \langle \lambda \rangle \cdot \uparrow_i x) : \gamma, \uparrow_i \alpha_1, \dots, \uparrow_i \alpha_k} \text{merge1}_{DG} \\
\frac{(\mathbf{s}, x) : = f\gamma, \alpha_1, \dots, \alpha_k \quad (\mathbf{t}, y) \cdot f, \iota_1, \dots, \iota_l}{(\mathbf{s} \cup \uparrow_i \mathbf{t}, \uparrow_i y \cdot x) : \gamma, \alpha_1, \dots, \alpha_k, \uparrow_i \iota_1, \dots, \uparrow_i \iota_l} \text{merge2}_{DG} \\
\frac{(\mathbf{s}, x) \cdot = f\gamma, \alpha_1, \dots, \alpha_k \quad (\mathbf{t}, y) \cdot f\delta, \iota_1, \dots, \iota_l}{(\mathbf{s}, x) : \gamma, \alpha_1, \dots, \alpha_k, (\uparrow_i \mathbf{t}, \uparrow_i y) : \delta, \uparrow_i \iota_1, \dots, \uparrow_i \iota_l} \text{merge3}_{DG}
\end{array}$$

where $i = \text{next}((\mathbf{s}, x) \cdot = f\gamma, \alpha_1, \dots, \alpha_k)$

Table 2: Merge in terms of dependency trees

3.1 Dependency trees

DG is typically discussed in terms of directed dependency *graphs*. However, the directed nature of dependency arrows and the single-headed condition (Nivre, 2006) allow these graphs to also be viewed as *trees*. We define dependency trees in terms of their nodes, with each node in a dependency tree labeled by an *address*, a sequence of positive integers. We write λ for the empty sequence of integers. Letters u, v, w are variables for addresses, \mathbf{s}, \mathbf{t} are variables for sets of addresses, and x, y are variables for sequences of addresses. If u and v are addresses, then the concatenation of the two is as well, denoted by uv . Given an address u and a set of addresses \mathbf{s} , we write $\uparrow_u \mathbf{s}$ for the set $\{uv \mid v \in \mathbf{s}\}$. Given an address u and a sequence of addresses $x = v_1, \dots, v_n$, we write $\uparrow_u x$ for the sequence uv_1, \dots, uv_n . $\uparrow_u \mathbf{s}$ is a *set* of addresses, whereas $\uparrow_u x$ is a *sequence* of addresses.

A *tree domain* is a set \mathbf{t} of addresses such that, for each address u and each integer $i \in \mathbb{N}$, if $ui \in \mathbf{t}$, then $u \in \mathbf{t}$ (prefix-closed), and $uj \in \mathbf{t}$ for all $1 \leq j \leq i$ (left-sibling closed). A *linearization* of a finite set S is a sequence of elements of S in which each element occurs exactly once. For the purposes of this paper, a *dependency tree* is a pair (\mathbf{t}, x) , where \mathbf{t} is a tree domain, and x is a linearization of \mathbf{t} . A *segmented dependency tree* is a non-empty sequence $(\mathbf{s}_1, x_1), \dots, (\mathbf{s}_n, x_n)$, where each \mathbf{s}_i is a set of addresses, each x_i is a linearization of \mathbf{s}_i , all sets \mathbf{s}_i are pairwise disjoint, and the union of the sets \mathbf{s}_i forms a tree domain. A pair (\mathbf{s}_i, x_i) is called a *component*, which corresponds to *chains* in Stabler and Keenan’s (2003) terminology.

An *expression* is a sequence of triples $(c_1, \tau_1, \gamma_1), \dots, (c_n, \tau_n, \gamma_n)$, where (c_1, \dots, c_n) is a segmented dependency tree, each τ_i is a type (lexical or derived), and each γ_i is a sequence of features. We write these triples as $c_i ::= \gamma_i$ (if the type is lexical), $c_i : \gamma_i$ (if the type is derived), or $c_i \cdot \gamma_i$ (if the type does not matter). We use the letters α and ι as variables for elements of an expression. Given an element $\alpha = ((\mathbf{s}, x), \tau, \gamma)$ and an address u , we write $\uparrow_u \alpha$ for the element $((\uparrow_u \mathbf{s}, \uparrow_u x), \tau, \gamma)$.

Given an expression d with associated tree domain \mathbf{t} , we write $\text{next}(d)$ for the minimal positive integer i such that $i \notin \mathbf{t}$.

3.2 Merge

Merge operations allow additional dependency structure to be added to an initially derived tree. These changes are recorded in the manipulation of the dependency tree addresses, as formalized in the previous section. Table 2 provides a dependency interpretation for each of the structure-building rules introduced in Table 1. The merge_{DG} functions create a dependency between two trees, such that the root of the left tree becomes the head of the root of the right tree, where left and right correspond to the trees in the rules. For example, in merge1_{DG} the $\uparrow_1 \mathbf{t}$ notation signifies that \mathbf{t} is now the first daughter of \mathbf{s} . Its components are similarly updated.

Applying the merge1_{DG} rule to a simple English grammar creates the dependency tree in Figure (1). Dependency relations between nodes are notated with solid arrows and node labels are notated with dotted lines; the dependency graphs shown in the figures are encoded

$$\frac{(\mathbf{s}, x) : +f\gamma, \alpha_1, \dots, \alpha_{i-1}, (\mathbf{t}, y) : -f, \alpha_{i+1}, \dots, \alpha_k}{(\mathbf{s} \cup \mathbf{t}, yx) : \gamma, \alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_k} \text{move1}_{DG}$$

$$\frac{\mathbf{s} \cdot +f\gamma, \alpha_1, \dots, \alpha_{i-1}, \mathbf{t} : -f\delta, \alpha_{i+1}, \dots, \alpha_k}{\mathbf{s} : \gamma, \alpha_1, \dots, \alpha_{i-1}, \mathbf{t} : \delta, \alpha_{i+1}, \dots, \alpha_k} \text{move2}_{DG}$$

Table 3: Move in terms of dependency trees

by the address sets described above in Table 2. A lexicon for these examples is provided in Figure 1(a).

As was mentioned above, the merge rules apply in different contexts depending on the tree types and number of features. **Merge1** can apply in Figure 1 because the selector tree **the** is simple and the selected tree **boat** does not have additional features δ . The entire derived tree forms a single component, denoted by the dashed box. This contrasts with an application of **merge3**_{DG}, where the second dependency tree has remaining features and becomes its own component in the dependency structure, shown in Figure 2(b).

The rules given in Table 2 are a deduction system for expressions—sequences of triples whose first components together define a segmented dependency tree. Each component, spanning any number of words, has a feature-sequence associated with it. **Merge3** introduces new, unlinearized components into the derivation. These components are unordered with respect to their parents, though the words within the components are ordered. **Merge2** contrasts with **merge1** in the linearized order of the nodes: in this case, the right tree is ordered before the left tree and its children, as in Figure 2(a).

3.3 Move

The move operation does not create or destroy dependencies in the derived tree. It re-orders the nodes, and reduces the number of components in the tree by one. Table 3 defines these rules in terms of dependency trees.

Figure 3 demonstrates the **move1**_{DG} operation on a simple structure. Not only has the **where** node been reordered to the front of the tree, but it also has become part of the ϵ node’s component.

It is important to note that **move2**_{DG} does not change the dependency structure or linearization of the tree. **Move2**_{DG} applies when the licensee component has additional features which will require further movements. It simply deletes the licensor and licensee features. Figure 4 shows how the additional licensee feature $-t$ remains after **move2**_{DG} applies.

Following Stabler (1997) and Stabler & Keenan (2003), we restrict movement with the Shortest Move Condition (SMC), defined in (1).

- (1) None of $\alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_k$ has $-f$ as its first feature.

Adoption of the SMC guarantees a version of MGs that are weakly equivalent to LCFRS (Gärtner & Michaelis, 2007).

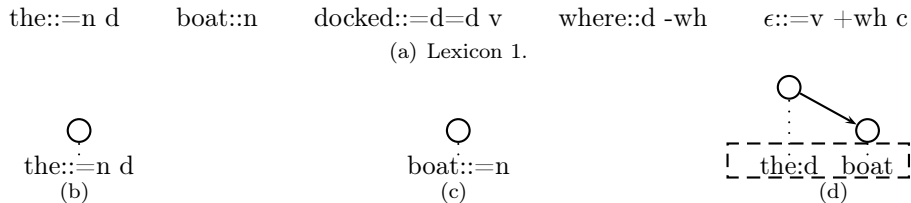


Figure 1: **merge1**_{DG} applies to two simple dependency trees.

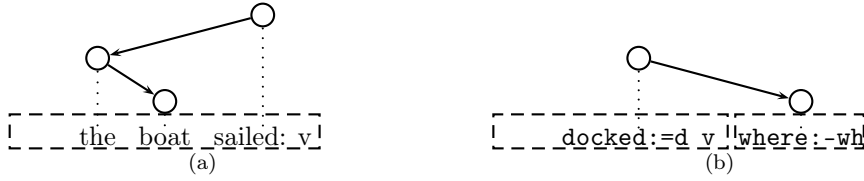


Figure 2: merge_{2DG} and merge_{3DG}.

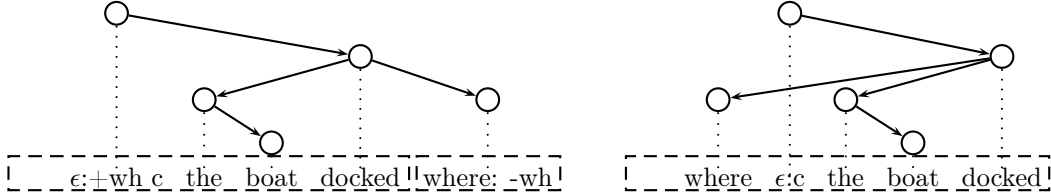


Figure 3: move_{1DG}.

The rules above derive connected dependency structures. This is easily shown by induction on the derived structure: Single-node trees (i.e., simple lexical items) vacuously satisfy connectedness. All merge rules create dependencies between trees, and movements do not destroy any already-created dependencies. Therefore, a dependency structure at any derivation step will be connected.

Provided that every expression in the lexicon has exactly one base feature, the dependency trees derived from MGs will not contain any multi-headed nodes (i.e., nodes with multiple parents). Such a “single-headedness” proof follows straightforwardly from two lemmas concerning the role of base features in expressions $E = (c_1, \tau_1, \gamma_1), \dots, (c_n, \tau_n, \gamma_n)$. Lemma 1 asserts the unique existence of a base feature f in the first feature sequence γ_1 . Lemma 2 denies the existence of any base features in later components.

The dependency structures derived at intermediate steps will not necessarily be totally ordered. Because of the merge₃ and move₂ rules, components can be introduced into the dependency structure that have not yet moved to their final order in the structure. However, the usual notion of start category (Stabler & Keenan, 2003) in MGs is a single base feature. This implies that in complete derivations all licensee feature have been checked. This implication also guarantees that dependency trees derived using the system in Tables 2 and 3 are totally-ordered.

4 Minimalist Grammars and block degree

Projectivity is a constraint on dependency structures that requires subtrees to span intervals. Kuhlmann & Nivre (2006) define an interval as the set $[i, j] := \{k \in V \mid i \leq k \text{ and } k \leq j\}$, where i and j are endpoints. V is the set of nodes, as defined in Section 3.1. Non-projective structures violate this constraint. In terms of intervals, the node labeled **docked** in Figure 3 spans two intervals in terms of order: its child spans interval 0 and the node and its other

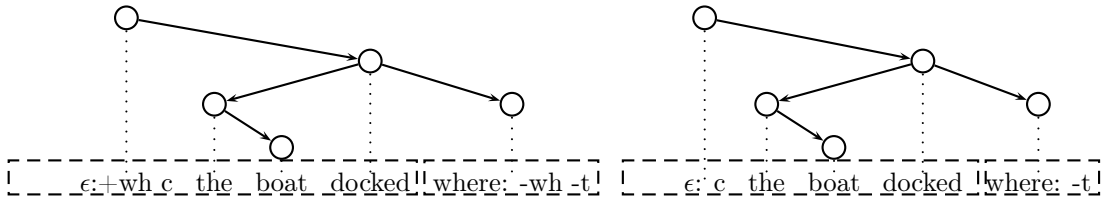


Figure 4: move_{2DG}.

children span intervals 2-4.

Following Kuhlmann (2007) we use the notion of block degrees to characterize non-projective structures. A tree’s block degree is the maximum number of intervals each of its subtrees span. For Figure 3 this would be two, as each of the intervals of the node labeled *docked* forms a block. We use shaded boxes to notate node blocks, as in Figure 5.

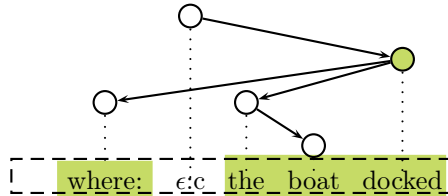


Figure 5: The block degree of this structure is 2.

By construction, merge_{DG} always forms dependency relations between the roots of subtrees. This means that all nodes in the resulting expression are part of the same interval. Move1_{DG} , however, has the potential to create non-projective structures: Constituents can move away from the interval that the parent node spans, creating a separate constituent block in terms of intervals, as demonstrated by Figure 5.

Because only movements can cause non-projectivity, and because all movements are triggered in the MG framework by a licenser and licensee pair, the block degree of the derived tree is bounded by the number of licensees. This number has previously been identified as an upper bound on the complexity of MGs (Michaelis, 1998; Harkema, 2001). The coincidence of this result follows from work by Satta (1992), who attributed the increased parsing complexity of LCFRS to non-projectivity (Kuhlmann, 2007).

5 Minimalist Grammars and nestedness

A further constraint on the class of dependency structures is wellnestedness (Kuhlmann, 2007). Wellnested structures prohibit the “crossing” of disjoint subtrees in terms of intervals. Any structure that is not wellnested is said to be illnested. Illnested structures like the one in Figure 6(b) allow for the crossing of disjoint subtrees. Kuhlmann (2007) demonstrates that grammars that can derive illnested structures are more powerful than grammars that do not, and lead to higher complexity in terms of parsing. We prove that MGs are able to derive illnested structures by example: The grammar in Figure 6(a) derives the illnested English structure in Figure 6(b).

Not all mildly context-sensitive formalisms can derive illnested structures. For example, TAGs can only generate wellnested structures with a block-degree of at most two (Kuhlmann, 2007). However, MGs can derive structures with higher block degrees, and there is the potential for illnested structures. This allows MGs to generate the same string languages as LCFRS, which can also generate illnested structures (Seki et al., 1991).

The illnested structure in Figure 6(b) is also interesting from a linguistic perspective. It represents a case of noun-complement clause extraposition (Guéron & May, 1984), where the complement **on the issue** is extraposed from the determiner phrase **the hearing**. The adverb **today** is also extraposed from the verb phrase **is scheduled** in this sentence, leading to the illnested final structure. Several analyses of extraposition are put forth in the literature, but here we choose Kayne’s (1994) “stranding analysis”, where a series of leftward movements leads to modifier stranding. These movements are each motivated by empty “functional” categories that could be overt in other possible human languages. In this lexicon, first the adverb is moved by the licenser $+w$ (Figure 7(a)), followed by the prepositional phrase, the verb phrase, and finally the noun phrase, as in Figures 7(b) through 7(e).

The analysis of the illnested structure in Figure 6(b) in terms of extraposition provides a first step towards an understanding of the linguistic relevance of illnested structures. This is

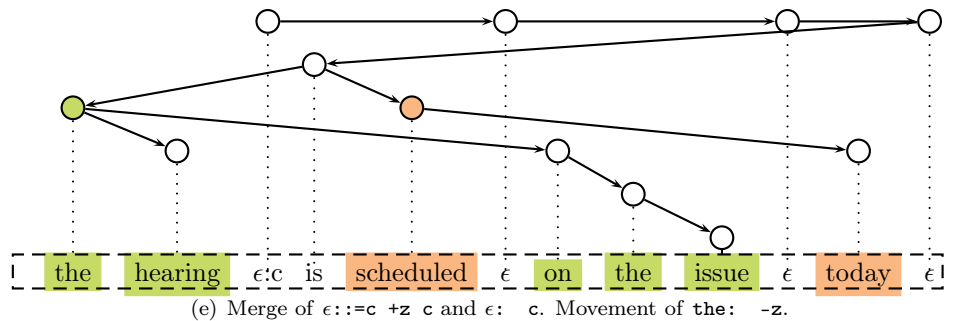
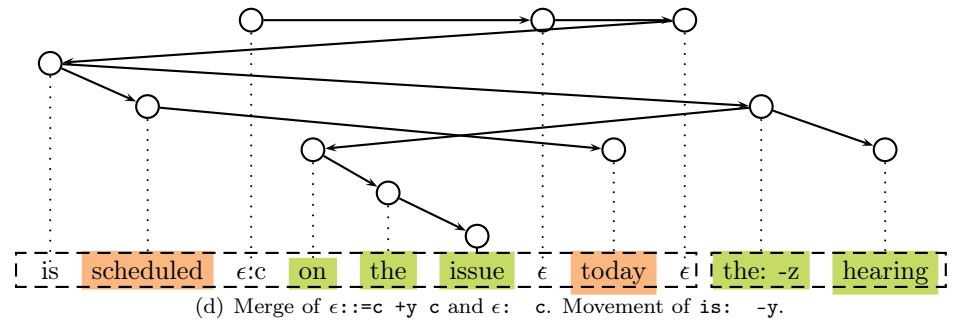
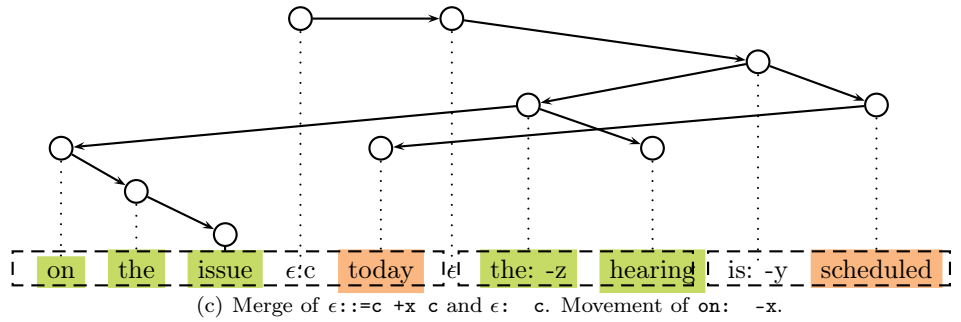
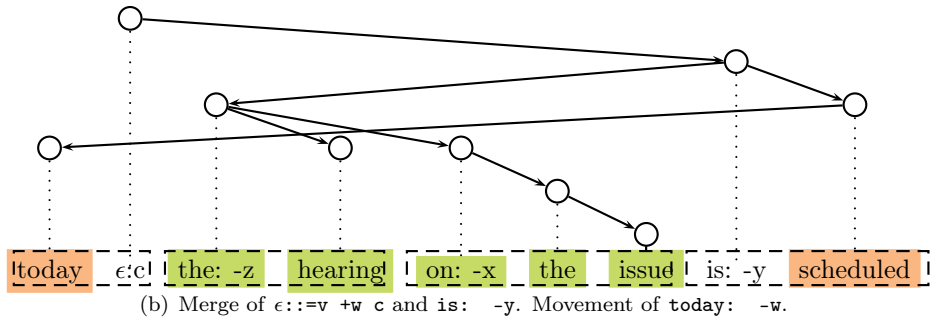
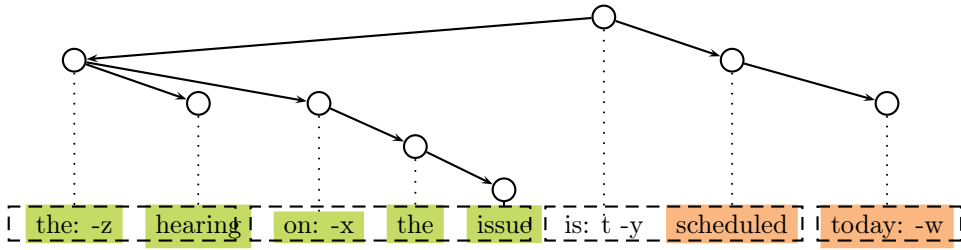


Figure 7: Derivation of illnested English structure.

References

Chomsky, N. (1995). *The minimalist program*. Boston, MA: MIT Press.

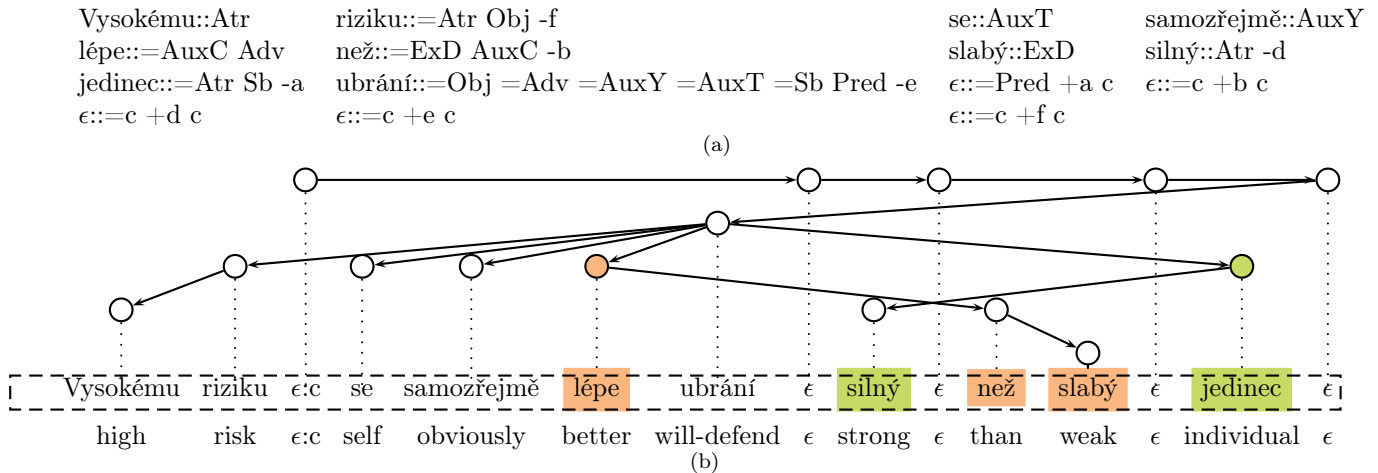


Figure 8: An illnested Czech example from the Prague Dependency Treebank.

Gärtner, H.-M., & Michaelis, J. (2007). Some remarks on locality conditions and Minimalist Grammars. In U. Sauerland & H.-M. Gärtner (Eds.), *Interfaces + recursion = language?* (pp. 161–195). Berlin: Mouton de Gruyter.

Goldberg, A. (2006). *Constructions at work: The nature of generalization in language*. New York, NY: Oxford University Press.

Guéron, J., & May, R. (1984). Extraposition and logical form. *Linguistic Inquiry*, 15, 1–32.

Hajič, J., Panevová, J., Hajičová, E., Sgall, P., Pajas, P., Štěpánek, J., et al. (2000). *Prague dependency treebank 2.0*.

Harkema, H. (2001). A characterization of minimalist languages. In *Proceedings of Logical Aspects of Computational Linguistics (LACL 2001)*.

Kayne, R. S. (1994). *The antisymmetry of syntax*. MIT Press.

Kuhlmann, M. (2007). *Dependency structures and lexicalized grammars*. Unpublished doctoral dissertation, Universität des Saarlandes.

Kuhlmann, M., & Nivre, J. (2006). Mildly non-projective dependency structures. In *Proceedings of the COLING/ACL 2006* (p. 507–514).

Michaelis, J. (1998). Derivational minimalism is mildly context-sensitive. In *Proceedings of Logical Aspects of Computational Linguistics (LACL 1998)*.

Michaelis, J. (2001). *On formal properties of Minimalist Grammars*. Potsdam: Universitätsbibliothek Publikationsstelle.

Nivre, J. (2006). *Inductive dependency parsing*. New York, NY: Springer.

Satta, G. (1992). Recognition of linear context-free rewriting systems. In *Proceedings of the Association for Computational Linguistics (ACL)* (pp. 89–95).

Seki, H., Matsumura, T., Fujii, M., & Kasami, T. (1991). On multiple context-free grammars. *Theoretical Computer Science*, 88(2), 191–229.

Stabler, E. P. (1997). Derivational minimalism. In *Proceedings of Logical Aspects of Computational Linguistics (LACL 1996)* (pp. 68–95).

Stabler, E. P., & Keenan, E. (2003). Structural similarity within and among languages. *Theoretical Computer Science*, 293(2), 345–363.

Tesnière, L. (1959). *Éléments de syntaxe structurale*. Editions Klincksiek.

Vijay-shanker, K., Weir, D. J., & Joshi, A. K. (1987). Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the Association for Computational Linguistics (ACL)* (pp. 104–111).