The computational unity of Merge and Move

Thomas Graf

mail@thomasgraf.net

Department of Linguistics

Stony Brook University

Stony Brook, NY 11794, USA

Abstract

Based on a formal analysis of the operations Merge and Move, I provide a computational answer to the question why Move might be an integral part of language. The answer is rooted in the framework of subregular complexity, which reveals that Merge is most succinctly analyzed in terms of the formal class TSL. Any cognitive device that can handle this level of complexity also possesses sufficient resources for Move. In fact, Merge and Move are remarkably similar instances of TSL. Consequently, Move has little computational or conceptual cost attached to it and comes essentially for free in any grammar that expresses Merge as compactly as possible.

Keywords: computational syntax, Minimalist grammars, subregular complexity, Merge, Move

1 Introduction

One of the central evolutionary questions posed by Minimalism is the origin of its two fundamental operations Merge and Move. Merge represents the ability to build larger structures from smaller ones and thus is indispensable for language. Move captures the displacement property, i.e. that parts of a sentence are sometimes pronounced in a position that is different from where they are interpreted. Chomsky (2004) reduces Move to Merge by defining it as the process of merging a structure S with a proper subpart P of S. In this paper, I argue that formal language theory allows us to sharpen this idea by building on existing results rooted in *Minimalist grammars* (MGs; Stabler 1997, 2011) and *subregular complexity* (see Heinz 2018 and references therein). I make three central claims: I) while Merge falls into the class of *strictly local* dependencies (SL), a more succinct and elegant picture emerges when Merge is viewed as a *tier-based strictly local* dependency (TSL); II) any cognitive device that handles Merge in terms of TSL also has the means to handle Move, so that the latter comes for free; and III) more specifically, Move and Merge are exactly analogous from this subregular perspective and may be regarded as one and the same operation.

The paper works its way towards this conclusion as follows: My vantage point is the result in Graf (2012) that Merge is SL in MGs (Sec. 2). The result uses a particularly fertile way to analyze syntactic operations: the structure-building operation Merge is converted into a conjunction of constraints on MG derivation trees, and each constraint is in turn equated with a set of well-formed derivation trees in order to measure its subregular complexity. While the constraints regulating Merge are all SL, they also lack succinctness and generality. The larger and more complex the lexicon, the more verbose the SL description of Merge becomes. If one wants a compact description of Merge that is largely independent of lexicon size, one has to take a step up to the subregular class TSL (Sec. 3).

Curiously, the TSL view of Merge exactly matches the TSL analysis of Move in the subregular literature. One might speculate, then, that evolutionary pressures to reduce memory usage caused a shift from SL Merge to TSL Merge, which resulted in a cognitive environment in which Move comes for free — a computational counterpart to Chomsky's reduction of Move to Merge.

2 Complexity of Merge

Let us start with an accessible summary of the finding in Graf (2012) that Merge is an SL dependency. Complexity claims of this kind always presuppose a computational model of Merge, which is provided by MGs. MGs are closely modeled after Minimalist syntax, although they differ in some respects. I will point out such differences whenever they are crucial for the results of this paper. This will be rare, though, as the core insights apply to any variant of Minimalism that adopts some version of subcategorization and feature-driven movement (as demonstrated in Graf 2017, most syntactic constraints can be replaced with features, so that even variants with a free Merge/Move operation limited by interface constraints is not obviously beyond the scope of this paper's argument).

I first discuss how Merge works in MGs in general and how derivation trees provide a linguistically faithful representation of syntactic structure (Sec. 2.1). This also allows for a reanalysis of Merge as a bundle of constraints on derivation trees (Sec. 2.2). I then explain why Merge is a locally bounded dependency and thus belongs to the very simple class SL (Sec. 2.3).

2.1 Merge in Minimalist grammars

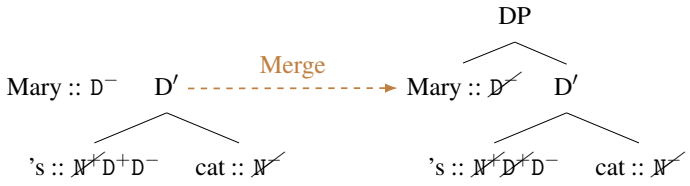
Following MG tradition, I assume that Merge does not apply freely but is mediated by a feature-driven subcategorization mechanism. That is to say, every lexical item (LI) has a

category feature $\texttt{X}^-$ and possibly one or more selector features $\texttt{Y}^+$, $\texttt{Z}^+$, and so on, that encode what arguments the LI requires. These category and selector features fully control the application of Merge.

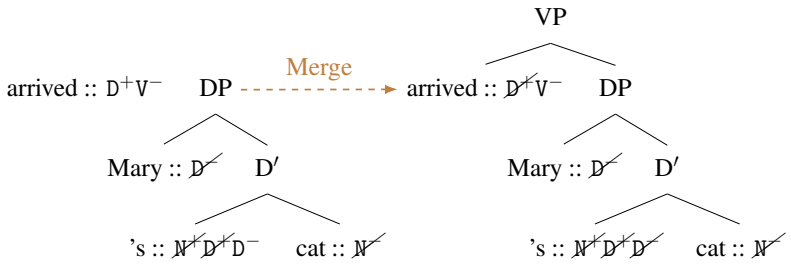
For example, the noun *cat* has only one feature, the category feature $\texttt{N}^-$. Therefore it can be selected by any LI that is looking for a noun, but it cannot take any arguments of its own. The determiner *a*, on the other hand, has the selector feature $\texttt{N}^+$ and the category feature $\texttt{D}^-$. More precisely, *a* carries the feature string $\texttt{N}^+\ \texttt{D}^-$ — the order of the features indicates that the determiner first has to merge with a noun before it can merge with an LI that is looking for a DP. The same logic dictates that the feature specification of the possessive marker *'s* is $\texttt{N}^+\ \texttt{D}^+\ \texttt{D}^-$ as it first merges with the possessee NP, then with the possessor DP, and only then can it act as a DP and merge with an appropriate selector, e.g. the unaccusative verb *arrive*. The corresponding sequence of Merge steps is depicted in (1) below with checked features crossed out, and with the notation $\alpha :: \beta$ to denote an LI with phonetic exponent α and feature string β.

(1) a.

b.



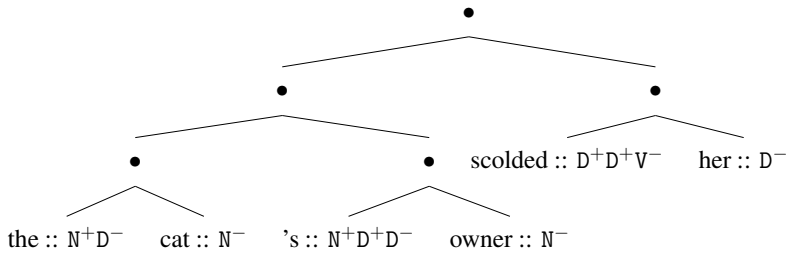
c.



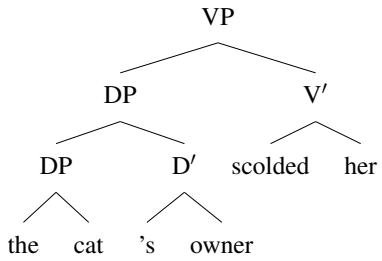
While feature ordering is never explicitly assumed in Minimalist syntax, there is an implicit consensus that whatever mediates selection exercises tight control over the order of arguments. This is why, say, *v* selects a VP as its complement and a subject DP as its specifier, rather than the other way around. The ordered Merge features of MGs thus are closely in line with linguistic practice, despite initial appearance to the contrary. For each LI, its string of selector and category features describes exactly what Merge steps an LI will partake in, similar to any other syntactic theory of subcategorization.

A sequence of Merge steps can be represented more succinctly as a *derivation tree* as in (2a), with the corresponding phrase structure tree shown in (2b). Each Merge step of the derivation is represented by an interior node labeled with •. For sake of succinctness, I assume that the subject is selected by V instead of *v*.

(2) a. **Derivation tree**



b. **Corresponding phrase structure tree**



Derivation trees provide a more abstract description of syntactic structure that focuses on the grammatical operations rather than their output, the *derived structures*. In Minimalist terminology, they are a direct representation of I-language operations, not objects of E-language. A derivation tree acts as a common blueprint for all of the following: a

canonical X′-tree, the compacted X′-tree in (2b), a bare phrase structure set, a PF-structure with prosodic modifications in the spirit of Richards (2016), a logical form, or simply the output string. Each one of these is produced from the same derivation tree. For this reason, derivation trees provide a unified representation format and are the ideal measuring rod for the complexity of the grammar's operations *modulo* differences in output representations.[1]

The remainder of this paper operates under the assumption that derivation trees are an abstract representation of the actual computations carried out by syntax. Consequently, the complexity of derivation trees is indicative of the complexity of syntactic computations. The next section explains how this perspective allows us to reinterpret Minimalist operations as constraints on derivation trees, which in turn makes it possible to measure the complexity of syntactic operations in terms of the complexity of the corresponding constraints on derivation trees.

2.2 Merge as a constraint on derivation trees

As discussed at the beginning of Sec. 2.1, Merge is a feature-triggered operation in MGs. So a computational system that has to correctly apply Merge must ensure that no requirements of the feature calculus are violated. For Merge, this involves two factors: Every selector feature must be checked against a matching category, and every category feature must be checked against a matching selector feature.

We can reduce these matching conditions to constraints on the shape of derivation trees. We will employ a specific procedure to asymmetrically connect features on LIs to interior nodes in the derivation tree (i.e. specific syntactic operations). In anticipation of the discussion of movement in Sec. 4, I use the terms *positive feature* and *negative feature* to refer to any feature with a superscripted plus or a superscripted minus, respectively. In an

[1] Derivation trees also satisfy many Minimalist desiderata for syntactic representations, in particular the Extension Condition, the Inclusiveness Condition, and lack of linear order.

MG without movement, selector features are the only positive features in the grammar, and category features are the only negative features.

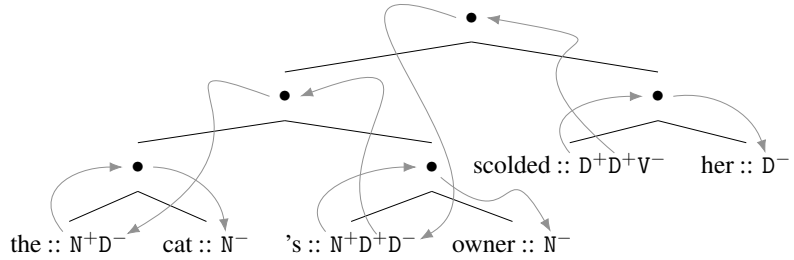
(3) **Two tree-geometric principles for connecting features and Merge nodes**

a. For every LI, its *i*-th positive feature $\texttt{F}^{+}$ is connected to the *i*-th node above the LI (if such a node exists). We also say that the node is *hosted* by $\texttt{F}^{+}$ and, by extension, the LI carrying said feature.

b. The *D[erivational]-root* of an LI *l* is the highest node hosted by *l* — if no such node exists, it is *l* itself. Suppose *l* carries some negative feature $\texttt{F}^{-}$. Then an interior node *m* is an F-*occurrence* of *l* iff *m* is the lowest node such that I) *m* properly dominates the D-root of *l*, and II) *m* is hosted by a matching positive feature $\texttt{F}^{+}$. Every F-occurrence of an LI is connected to the negative feature $\texttt{F}^{-}$ of the LI.

Let us annotate an example derivation with arrows so as to make these connections between features and nodes fully explicit: each positive feature has an arrow going to the node it hosts, and each interior node *n* is connected to the negative feature $\texttt{F}^{+}$ on LI *l* that makes *n* an F-occurrence of *l*. These arrows are an expository device to simplify the discussion, not part of the actual derivation tree.

(4) **Derivation tree with arrows as visual aid**



The feature calculus driving Merge is equivalent to three constraints on the arrangement of arrows.

(5) **Merge constraints on derivation trees**

a. *Single Head*

Every interior node is hosted by exactly one feature (i.e. it has exactly one incoming arrow).

b. *Full Projection*

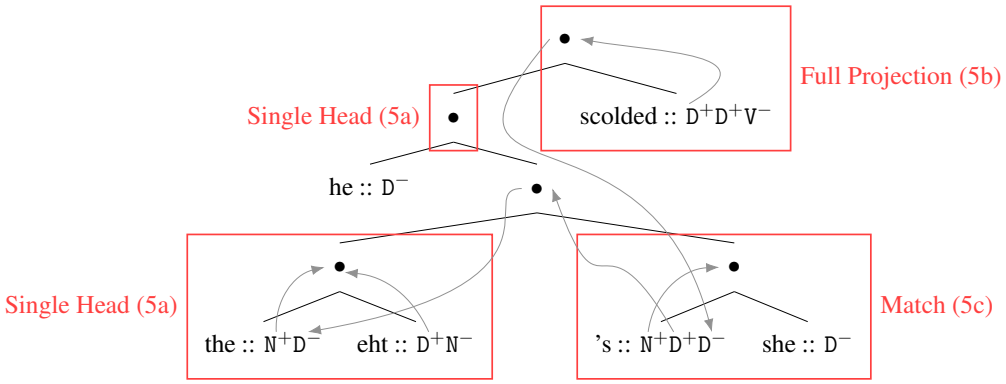
Every LI with exactly $n \geq 0$ positive features hosts exactly n interior nodes (i.e. it has exactly n outgoing arrows).

c. *Match*

Let m be an interior node hosted by some positive feature $\mathtt{F}^+$ (i.e. it has an incoming arrow from some $\mathtt{F}^+$). Then m is an $\mathtt{F}$-occurrence of exactly one LI (i.e. m has exactly one outgoing arrow, which must end in some negative feature $\mathtt{F}^-$).

A derivation tree contains an illicit Merge application iff one of the constraints above is violated. The example below shows a derivation that does not obey the MG feature calculus, and how these violations correspond to illicit arrow configurations.

(6) **A derivation that violates every constraint in** (5)



The representational, constraint-based view of the operation Merge allows us to assess the complexity of Merge in terms of a specific formal problem. Let *Lex* be a set of LIs

annotated with selector and category features in the usual manner. It is usually assumed in the MG literature that *Lex* is finite (this is not at odds with the linguistic idea of an infinitely productive lexicon, which can be regarded as a finite lexicon with the ability to add new lexical items on the fly). Then there is a unique (and usually infinite) set of well-formed derivation trees that can be built from *Lex*. As long as *Lex* is finite, this set forms a *tree language*, just like a *string language* is a set of strings over a finite set of symbols. And just as with string languages, formal language theory provides ways of measuring the complexity of tree languages. With respect to Merge, the only criterion for well-formedness of a derivation tree is whether it obeys the constraints in (5). Therefore the complexity of Merge can be equated with the complexity of these derivation tree languages.

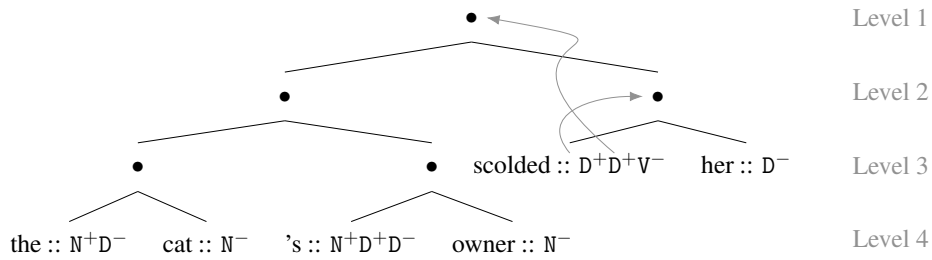
2.3 Merge is strictly local (SL)

Correctly employing Merge hinges on the ability to track how nodes in the derivation tree are related to each other, which we visualized with arrows in (4) and (6). The arrows are not part of the actual representation, they merely depict the inter-node dependencies that the computational system has to infer on its own. Still, the arrows make it easier for an external observer to analyze the difficulty of this task, and one particular property of these arrows will reveal Merge to be an exceedingly simple operation.

Upon closer inspection, it quickly becomes clear that given some lexicon *Lex* for a grammar that only uses Merge, there is no well-formed derivation tree for *Lex* such that any arrow in that derivation tree has a length that exceeds some fixed threshold k. Recall that every incoming arrow spans from an LI's i-th positive feature to the i-th Merge node above the LI. The length of such arrows thus is finitely bounded by the number of positive features an LI may carry. But since there are only finitely many distinct feature strings in *Lex*, and each feature string must be finite, there is some upper bound p such that no LI in *Lex* has

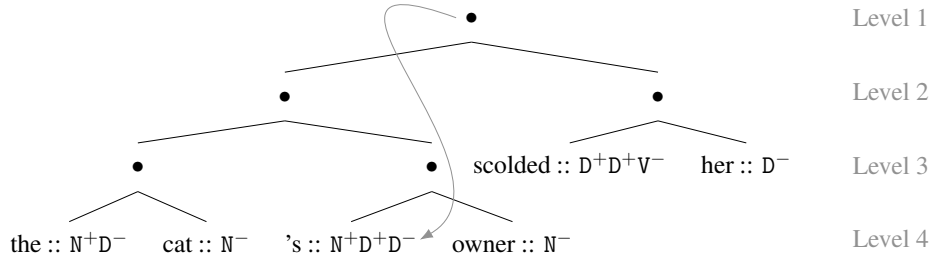
more than p positive features. But then an arrow from a selector feature to the corresponding Merge node never spans across more than $p+1$ "levels" in the tree, as is illustrated in (7).

(7) **An arrow from an LI with 2 selector features spans at most 3 levels**



This also implies an upper bound on the length of outgoing arrows, which span from a Merge node m to the category feature of the LI l that the Merge node is an occurrence of. In a well-formed derivation, it is always the case that m is the mother of l's D-root (one can show that whenever this is not the case, the derivation tree contains some other Merge node that violates one of the three conditions in (5)). But since an arrow from l's D-root to l never spans more than $p+1$ levels, the distance between l and the mother of the D-root is at most $p+2$ levels.

(8) **An arrow from a Merge node to an LI with 2 selector features spans 4 levels**



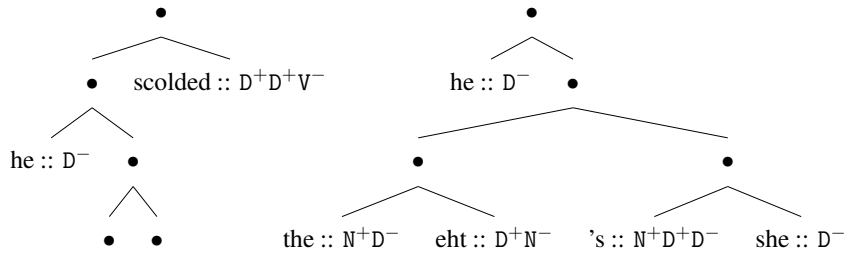
Arrows, or rather the Merge dependencies that they visualize, thus never span more than $p+2$ levels in the derivation tree, giving us an upper bound of $k = p+2$ such that one never needs to consider more than k levels at once in order to detect an illicit configuration. In the terminology of formal language theory, this makes Merge a *strictly k-local* (SL-k)

dependency, where k depends on the maximum number of positive features a single LI may carry.

A dependency is SL-k iff one can determine the well-formedness of the whole structure merely by investigating all its substructures of size k. The class SL has come to prominence in computational phonology (see Heinz 2018, Chandlee this volume, and references therein), where k measures the number of adjacent segments. A staggering number of phonotactic phenomena are strictly local, which is noteworthy because this represents one of the weakest known classes in formal language theory. For example, intervocalic voicing, when construed as a phonotactic constraint on surface forms, is SL-3 because it suffices to ensure that no sequences of three adjacent segments is of the form V[−voice]V. Hence Northern Italian *azola* is well-formed because its substructures of size 3 are *azo*, *zol*, and *ola*, none of which match the illicit pattern V[−voice]V. A putative form *asola*, on the other hand, would contain the illicit *aso*, so that the whole string is ill-formed because one of its substructures of size 3 does not obey intervocalic voicing. However, the morphologically complex *asociale* would still be permitted if one treats the morpheme boundary as a segment in its own right, so that the underlying structure is actually *a+sociale*, whose size-3 substructures are *a+s*, *+so*, *soc*, *oci*, *cia*, and *ale*, none of which violate intervocalic voicing.

The very same logic applies for SL-k over trees, except that k now indicates each substructure's number of levels instead of the number of segments/nodes. The illicit derivation tree in (6), for instance, contains no LI with more than 2 positive features, and every violation is indeed detectable within $2+2=4$ levels.

(9) **Both size-4 substructures of** (6) **are illicit**



The left substructure is necessarily illicit because I) we know that no LI has more than 2 selector features, and II) the Merge node above *he* is neither the mother of an LI with at least one selector feature, nor the mother of the mother of an LI with two selector features. The violations of the Single Head condition (5a) and the Match condition (5c) are just as readily apparent in the substructure on the right.[2] In a grammar that only uses Merge and where no LI has more than p positive features, determining the well-formedness of a derivation tree reduces to determining the well-formedness of its substructures with $p+2$ levels. This reduction step from a structure of arbitrary size to substructures of fixed size is what makes SL one of the simplest classes in formal language theory.

Rogers and Pullum (2011) argue that it also makes SL maximally simple from a cognitive perspective, in at least two respects. First, only a single size-k substructure needs to be worked on during any given point of the computation, which reduces working memory requirements. For our earlier example of intervocalic voicing, one need not work on the whole string *asola* at once, it is sufficient to work through it from left-to-right considering only three consecutive symbols at any given moment. Similarly, a derivation tree can be evaluated piece by piece by only considering one substructure of depth k at a time.

Second, determining the well-formedness of a size-k substructure is maximally simple. For any integer k, there are only finitely many structures of size k. Hence one can simply memorize all well-formed size-k substructures. For instance, intervocalic voicing can be

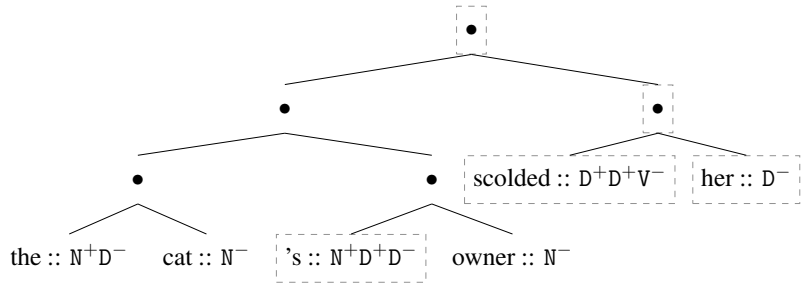
[2]The violation of the Full Projection constraint (5b) can be detected because SL can distinguish nodes at the edge of the full structure — the first and last segment in a string, the root and the leaves of a tree — from nodes inside the structure. In the case at hand, *scolded* has two selector features yet its mother is also the root of the whole derivation tree, precluding the existence of the required second Merge node.

mastered by listing all trigrams that do not violate it. There is no need for elaborate inference rules, list look-up is fully sufficient. Of course an abstract pattern like V[−voice]V will often provide a more succinct description, but it also requires a mechanism to correctly compare substructures against this template. In terms of the bare cognitive minimum that must be in place for SL-dependencies to emerge, all one needs is a small working memory that can hold the relevant substructures, and some long-term storage that lists the allowed substructures. In other words, the fact that Merge is an SL dependency makes it a very natural evolutionary starting point for syntax. But as I will argue next, that SL is a natural starting point for Merge does not entail that SL is also a natural end point for Merge.

3 Merge as a tier-based strictly local dependency

Merge — when construed as a collection of constraints on derivation trees — is computationally simple by virtue of being an SL-*k* dependency, but that does not guarantee that it is simple from a cognitive perspective. The set of well-formed size-*k* substructures is very large for any reasonably complex grammar, making brute-force memorization an unlikely evolutionary scenario. Most of that memorization is a waste because only a few nodes in any given substructure actually matter for a specific application of Merge.

(10) **Only the boxed nodes matter for the features on *scolded***



Any reasonably large grammar will contains thousands of variants of the substructure above depending on how on instantiates the nodes that are not boxed. None of these changes affect

whether Merge was applied correctly with respect to *scolded*, yet the memorization approach to SL dependencies would have to store each one of them.

If there are evolutionary pressures to minimize cognitive resource load, then the brute-force memorization approach is expected to become disfavored as the lexicon grows in size; alternative, more succinct methods should emerge. I present two findings along those lines. First, the constraints Single Head and Full Projection can be lexicalized into *slices* (Graf 2012; Kobele 2011), which are a metaphor for a system that memorizes not well-formed substructures but what operations have to follow the introduction of a specific LI (Sec. 3.1). Second, the Match constraint allows for a very simple description in terms of tree tiers, which represent the ability to consider only those parts of a derivation that matter for a specific type of Merge operation (Sec. 3.2).

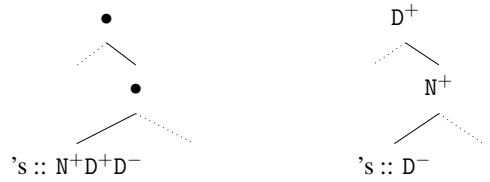
3.1 Lexicalizing Single Head and Full Projection

The discussion of Merge as a collection of constraints on derivation trees implicitly assumed a generate-and-filter approach: we consider any random tree where nodes denote LIs or Merge steps, and then use the three constraints Single Head, Full Projection, and Match to reduce this set to only those trees that encode well-formed derivations with respect to Merge (not entirely unlike the free Merge system in recent Minimalist proposals). This is a standard approach for studying complexity, but there are alternatives that shift some of the workload from constraints into the generation itself. In the case of MG derivation trees, there is a cognitively plausible scenario for this that also allows us to do away with Single Head and Full Projection by lexicalizing them.

In MGs, it is already assumed that every LI comes with a linear sequence of positive features that must be checked in this exact order. So if one adds, say, the LI scolded :: $\mathtt{D}^{+}\mathtt{D}^{+}\mathtt{V}^{-}$ to the derivation tree, the only way to obtain a licit derivation is if this is

followed up by two Merge steps. Suppose, then, that the cognitive system does not merely memorize an LI's feature string, but also the operations that necessarily follow it in every well-formed derivation. In formal terms, the lexicon no longer stores individual LIs, but derivational *slices* as in (11) on the left. But once the lexicon memorizes slices, then we might just as well remove positive features from the LI and put them directly into the slice, as in (11) on the right.

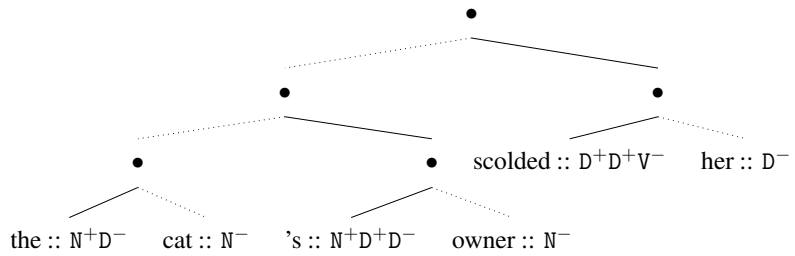
(11) **Two slice formats for the possessive marker**

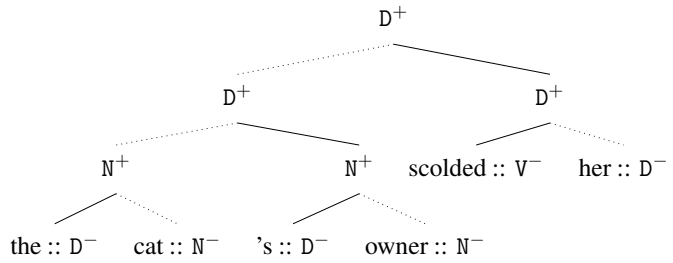


The slice on the left encodes that the possessive marker *'s* must always trigger two consecutive Merge steps due to its two selector features, whereas the more detailed one on the right stores the triggering feature instead of the type of operation (which is easily inferred from the feature).

A derivation tree is constructed by freely combining slices stored in the lexicon.

(12) **Derivation tree as combination of slices (with and without feature projections)**





Any such combination of slices is guaranteed to obey Full Projection and Single Head. Full Projection requires every positive feature to host a corresponding operation in the derivation. But this necessarily holds because slices are explicitly constructed by adding one interior node for each positive feature, and nodes cannot get lost when combining slices. Single Head, on the other hand, enforces that every interior node m is hosted by exactly one LI, which decomposes into two requirements: m is hosted by at least one LI, and m is hosted by at most one LI. The first is trivially satisfied because m is present in the tree iff it is present in the slice of some LI l. But then m is hosted by l because every interior node of the slice is tied to some positive feature of l. For the very same reason, there cannot be another LI l' such that m is hosted by both l and l'. Due to how slices are constructed, every LI hosts only the nodes in its own slice, and it is impossible for m to belong to both the slice of l and the slice of l'.

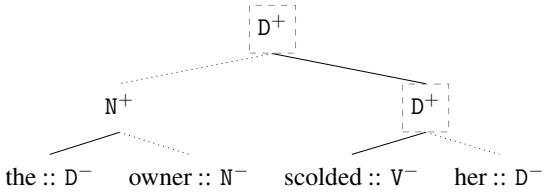
We see, then, that Full Projection and Single Head need not be enforced by considering a full substructure, we can filter out irrelevant parts of the structure and focus only on the nodes right above an LI. This can be thought of as memorizing what operations necessarily follow the insertion of any given LI, which we represent in formal terms as the storing and combining of slices. The specific feature that triggers each operation may be memorized, too, giving us a feature-annotated form of slices. Either way the resulting system is still SL: slices do not sneak in any additional power, they just provide a more succinct and elegant way to specify these SL dependencies. By contrast, we will see next that the Match constraint can also be given a more elegant description by filtering out irrelevant parts of the

structure, but in this case the filtering does increase the grammar's expressivity. However, this increase in expressivity is exactly what paves the way for the emergence of Move.

3.2 Merge tiers: The intuition

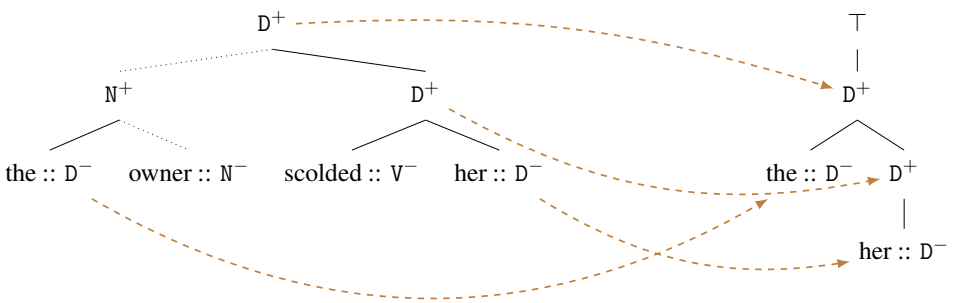
Consider the derivation tree in (12), which is assembled from feature-annotated slices. Suppose that we are interested in whether the two Merge steps in dashed boxes satisfy the Match constraint.

(13)



Since both Merge steps involve D+, the issue is whether each Merge node is an occurrence for an LI carrying D−, and how we could determine this. As a start, let us remove all nodes that are not D+ or an LI carrying D−, without altering the dominance relations between any of the remaining nodes. This is shown in (14) (dashed arrows connect nodes in the derivation tree to their counterpart in the truncated structure). For reasons that will become clear later on, we also add a special root marker ⊤ to the structure.

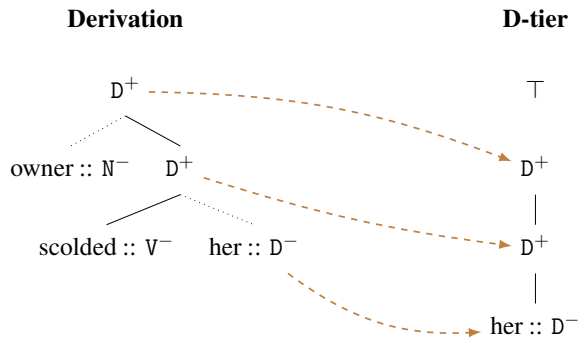
(14)



This truncated structure is called a *tree tier*, inspired by the notion of tiers in phonology. More precisely, we may call it a D-*tier* because it contains all nodes that are in some way related to a positive or negative D-feature.

On the D-tier in (14), each selector feature $\texttt{D}^+$ has exactly one LI with category feature $\texttt{D}^-$ among its daughters. Contrast this against an ill-formed version of (13), where the higher Merge step violates the Match constraint and we also find that the D-tier contains a $\texttt{D}^+$ without a suitable $\texttt{D}^-$ daughter.

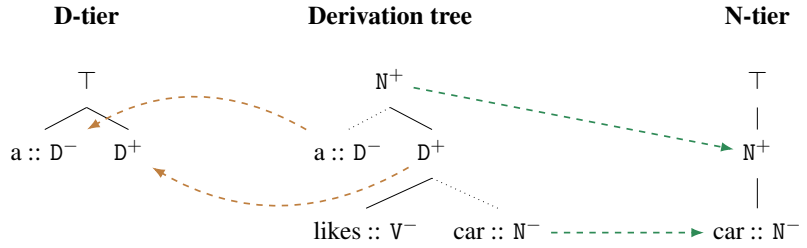
(15)



This suggests that a derivation tree contains a violation of the Match constraint iff there is some tier on which a positive feature does not have exactly one negative feature among its daughters.

This is indeed the case, although it is essential to construct and check every tier for every Merge feature. For example, the derivation tree in (16) contains a violation of the Match constraint with respect to the selector feature $\texttt{N}^+$ on *a*, yet the N-tier itself obeys the requirement that every $\texttt{N}^+$ has exactly one $\texttt{N}^-$ among its daughters. However, the violation of the Match constraint for $\texttt{N}^+$ on *a* also causes another Match violation for $\texttt{D}^+$ on *likes*, and this violation does show up on the D-tier. The D-tier also shows why each tier has a distinguished root marker $\top$: without it, tree tiers are not guaranteed to be trees, which would complicate the subsequent formalization in Sec. 3.3.

(16)



Example (16) illustrates a more general principle: even though not every Match violation is visible on a tier, every derivation tree that violates Match contains at last one Match violation that does show up on a tier. In light of this fact, we can replace the Match constraint with the two principles in (17).

(17) **F-tier projection**

The F-tier of a derivation tree contains all nodes labeled $\texttt{F}^+$ and all LIs that carry $\texttt{F}^-$. Dominance relations are carried over from the derivation tree. The tier has a unique root $\top$.

(18) **Match over tiers**

No F-tier may contain a node *m* labeled $\texttt{F}^+$ without exactly one LI among the daughters of *m*.

The constraint in (18) indirectly enforces the match condition on Merge as a condition on the shape of Merge tiers. If one uses slices without feature annotations, whose interior nodes are labeled •, then the conditions have to be stated slightly differently, but they follow the same principle of projecting both the relevant Merge nodes and the relevant LIs, and using mother-daughter configurations to ensure that every positive feature has a matching negative feature.[3]

While the preceding discussion does not qualify as a formal proof, it conveys the intuition how the Match constraint on Merge can be reduced to maximally local

[3]Some technical complications arise with LIs that carry both a category feature $\texttt{F}^-$ and a selector feature $\texttt{F}^+$, e.g. the possessive marker 's :: $\texttt{N}^+\texttt{D}^+\texttt{D}^-$. In this case, one has to consider a larger locality domain over the tier than just the mother-daughter configuration. But the very same issues can also arise with Move, so that they do not undermine the subregular parallels between Merge and Move.

dependencies between mothers and daughters, assuming that one can filter out intervening material that does not matter for this dependency. The next section refines the general idea by building on the subregular notion of *tier-based strictly local* (TSL) dependencies.

3.3 The tier-based strictly local model of Merge

Even though TSL was originally defined as an extension of SL over strings in order to model various phonological phenomena, e.g. long-distance harmony (Heinz et al. 2011; De Santo and Graf 2019), it can also be applied to trees (Graf and Kostyszyn 2021). The central idea is that a dependency is TSL iff it is SL over a tier. In the case of TSL over trees, this takes the form of two functions: one for constructing a tier, and another one for verifying its well-formedness in a strictly local manner.

(19) **Components of TSL definition for trees**

a. The *projection function* determines which nodes are ignored and which are added to the tier. In standard TSL, the projection function makes this decision based solely on the label of the node, but more powerful versions also allow local context to be taken into account (De Santo and Graf 2019).

b. Each tier comes with a *licensing function* that maps each node on the tier to a (possibly infinite) set L of permitted *daughter strings*. The tier is illicit if the node's string of daughters is not a member of L. Again this function may consider only the label of the node or take its structural context into account.

The approach from the previous section is easily recast in terms of these two components, although they take slightly different form depending on whether slices are feature-annotated. Due to space constraints, I only discuss the latter here (but a summary of the former is included in (20)). Suppose, then, that every interior node in the derivation is annotated with the positive feature (i.e. the selector feature) that hosts this node. In this case,

each F-tier uses a tier projection function that projects a node iff it is an LI carrying the negative feature $\texttt{F}^-$ or an interior node annotated with $\texttt{F}^+$. The licensing function then maps every $\texttt{F}^+$ to the string language $\texttt{F}^+_{\geq 0}\ \texttt{F}^-\ \texttt{F}^+_{\geq 0}$, where $\texttt{F}^-$ is a shorthand for any LI with category feature $\texttt{F}^-$, and $\texttt{F}^+_{\geq 0}$ denotes 0 or more iterations of $\texttt{F}^+$. This enforces that every Merge node must have exactly one LI among its daughters. LIs are not allowed to have any daughters, and consequently the licensing function maps each one of them to the empty set. The tier root $\top$, finally, varies between tiers. In most tiers, the set of daughter strings for $\top$ is $\texttt{F}^+_{\geq 0}$, which prevents LIs from occurring at the top of the tier. This is done so as to ensure that every LI has a Merge node as its tier mother, which in turn means that the LI gets selected during the derivation and has its category feature checked. However, there is at least one case where this is undesirable, namely for $\texttt{C}^-$ on the head of the matrix CP. This feature never gets checked because the matrix CP is not selected by anything else. For C-tiers, then, the daughter string of $\top$ is $\texttt{C}^+_{\geq 0}\ \texttt{C}^-\ \texttt{C}^+_{\geq 0}$, which guarantees the presence of exactly one C-head that is not selected by anything else. These definitions change only marginally if one uses slices without feature annotation, the key difference being that the projection function now has to inspect the local context to determine which feature a given Merge node is hosted by.

(20) **TSL-specification of Merge**

	With feature annotation	**Without feature annotation**
Projection	LI with $\texttt{F}^-$	LI with $\texttt{F}^-$
	$\texttt{F}^+$	$\bullet$ if hosted by $\texttt{F}^+$
Licensing	$\texttt{F}^+ \mapsto \texttt{F}^+_{\geq 0}\ \texttt{F}^-\ \texttt{F}^+_{\geq 0}$	$\bullet \mapsto \bullet_{\geq 0}\ \texttt{F}^-\ \bullet_{\geq 0}$
	$\top \mapsto \texttt{F}^+_{\geq 0}$	$\top \mapsto \bullet_{\geq 0}$
	$\top \mapsto \texttt{C}^+_{\geq 0}\ \texttt{C}^-\ \texttt{C}^+_{\geq 0}$ for C-tier	$\top \mapsto \bullet_{\geq 0}\ \texttt{C}^-\ \bullet_{\geq 0}$ for C-tier

Within the class of TSL dependencies over trees, Merge is surprisingly simple. The

pattern $\mathtt{F}^{+}_{\geq 0}\mathtt{F}^{-}\mathtt{F}^{+}_{\geq 0}$ (or equivalently, $\bullet_{\geq 0}\mathtt{F}^{-}\bullet_{\geq 0}$) used by the licensing function is particularly remarkable. First, it is TSL, too, albeit over strings instead of trees — take a string of the form $\mathtt{F}^{+}_{\geq 0}\mathtt{F}^{-}\mathtt{F}^{+}_{\geq 0}$, project a string tier that contains only LIs with $\mathtt{F}^{-}$, and then enforce the SL constraint that the tier has a length of 1. Second, $\mathtt{F}^{+}_{\geq 0}\mathtt{F}^{-}\mathtt{F}^{+}_{\geq 0}$ can be regarded as a syntactic counterpart to the phonological property *culminativity*, i.e. that every phonological word has exactly one primary stress. This corresponds to the formal language $\sigma_{\geq 0}\acute{\sigma}\sigma_{\geq 0}$, where σ denotes an unstressed syllable and $\acute{\sigma}$ a stressed one. The parallels between $\mathtt{F}^{+}_{\geq 0}\mathtt{F}^{-}\mathtt{F}^{+}_{\geq 0}$ and $\sigma_{\geq 0}\acute{\sigma}\sigma_{\geq 0}$ are evident. Perhaps, then, a putative evolutionary step from an SL-system of Merge to a TSL-system could have co-opted computational machinery that was already in use in phonology — or the other way round, the step to a more powerful Merge operation paved the way for more complex phenomena in phonology. This is highly speculative, of course, but it illustrates how a perspective informed by formal language theory helps identify novel connections between seemingly unrelated parts of linguistic cognition.

It is important to keep in mind, though, that these mathematical insights are necessarily abstractions and should not be interpreted too literally. In particular the notion of tiers should not be reified. Recall from Sec. 2.1 that derivation trees are a record of the computations carried out by the grammar. This seems at odds with the notion of tree tiers, for what could it possibly mean to project a tier from a computation? But this question assumes that tiers are an actual structure of some kind, when they are in fact a metaphor for what kind of memory and inference mechanisms have to be available to the cognitive system. The insight is not that syntax literally projects tiers from tree representations in order to determine if Merge is being used correctly. Rather, the SL-dependencies underlying Merge can be verified in a more succinct and elegant manner if the inference mechanisms can ignore irrelevant material. Recall that the SL-complexity of Merge depends on the maximum number p of positive features an LI may carry, which may vary across grammars. The TSL-account, on the other hand, is always one of mother-daughter

relations on tiers, irrespective of the value of p. TSL thus provides a unified description of Merge across all grammars, and since only very small chunks of structure need to be worked on at a time, it also has lower working memory requirements.

There are many reasons to believe, then, that Merge is best understood as a TSL dependency even though its complexity is just SL. From a linguistic perspective, it provides a unified description of Merge, the complexity of which is constant across grammars. From a cognitive perspective, the shift to a more sophisticated inference mechanism is welcome as it greatly reduces memory requirements. From an evolutionary perspective, TSL is appealing because it is not specific to syntax but also plays a role in phonology and possibly morphology (Aksënova et al. 2016), perhaps even semantics (Graf 2019). And as we will see next, it reveals a great degree of parallelism between Merge and Move.

4 Move is tier-based strictly local, too

The methods I used to show that Merge is TSL work just as well for Move. Given a basic understanding of Move in MGs and how it is characterized as a constraint on derivation trees (Sec. 4.1), it quickly becomes clear that these constraints are local over specific tiers, and that these tiers are but notational variants of the Merge tiers we just encountered (Sec. 4.2). The only significant difference between Merge and Move is how they affect the output structure, but even in that respect Move stays within the realm of TSL.

4.1 Move in MGs

The MG operation *Move* is modeled closely after the Minimalist notion of movement and is fully controlled by the feature calculus. Just like Merge is triggered by the presence of both a selector feature $\texttt{F}^{+}$ and a category feature $\texttt{F}^{-}$, Move requires a *licensor feature* $\texttt{f}^{+}$ and a *licensee feature* $\texttt{f}^{-}$. By convention, Move features are written in lower case like $\texttt{f}^{+}$ and $\texttt{f}^{-}$

to distinguish them from Merge features. Licensor features must always occur before an LI's category feature, which rules out countercyclic movement where a head attracts a mover after it has already been selected and thus had its category feature checked. Licensee features, on the other hand, always follow the category feature, so that a phrase cannot participate in any movement until it has been selected by some other head. Given these constraints on feature order, a well-formed derivation could contain the LI good :: $\texttt{D}^{+}\texttt{f}^{+}\texttt{A}^{-}\texttt{g}^{-}$, but not bad :: $\texttt{f}^{+}\texttt{A}^{-}\texttt{g}^{-}\texttt{D}^{+}$ or worse :: $\texttt{g}^{-}\texttt{D}^{+}\texttt{A}^{-}\texttt{f}^{+}$. Quite generally, all positive features must precede all negative features.

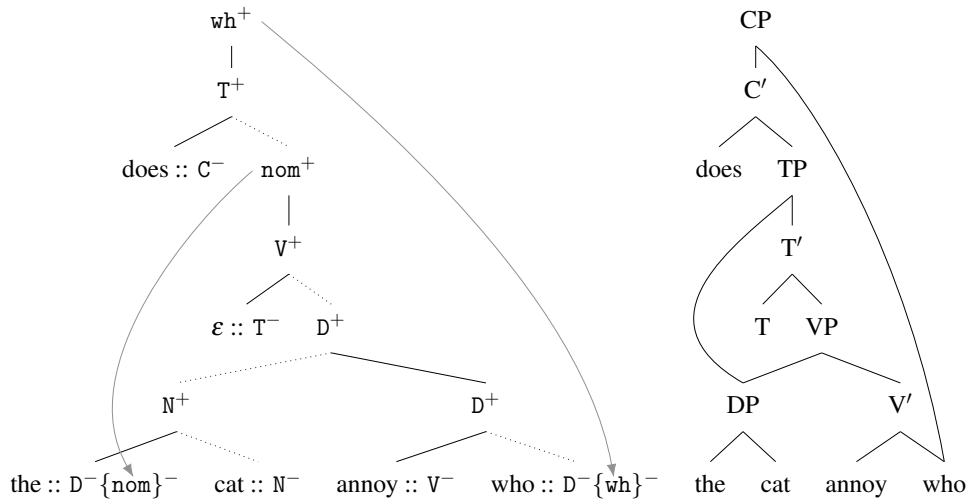
A single LI may have more than one licensee feature, for instance, $\texttt{nom}^{-}$ to undergo movement to subject position and $\texttt{wh}^{-}$ for wh-movement ($\texttt{nom}^{-}$ is an arbitrary feature name and should not be construed as a theoretical claim that subject movement is intrinsically tied to checking of nominative case). In standard MGs, all licensee features are linearly ordered, e.g. which :: $\texttt{N}^{+}\texttt{D}^{-}\texttt{nom}^{-}\texttt{wh}^{-}$. But this means that *which* does not count as a possible wh-mover until it has undergone subject movement. This has the side effect that another wh-phrase *w* should be able to wh-move across the wh-subject *s* as long as the wh-movement of *w* targets a position below the subject position of *s*. As this is not in line with standard Minimalist thinking, I will assume that licensee features are unordered, as in which :: $\texttt{N}^{+}\texttt{D}^{-}\{\texttt{nom},\texttt{wh}\}^{-}$. This does not affect the generative capacity of the formalism (Graf et al. 2016), but it is a prerequisite for the TSL account of Move (it is interesting that this point where MGs deviate from linguistic practice is exactly one where TSL argues against the former and for the latter).

With this system of unordered licensee features, each LI undergoes up to three consecutive stages in a derivation: I) selecting 0 or more arguments with selector features and attracting 0 or more movers with licensor features, and II) being selected by some head via its category feature, and III) undergoing 0 or more movement steps from its argument position to some higher landing site(s), where the LI always targets the closest possible

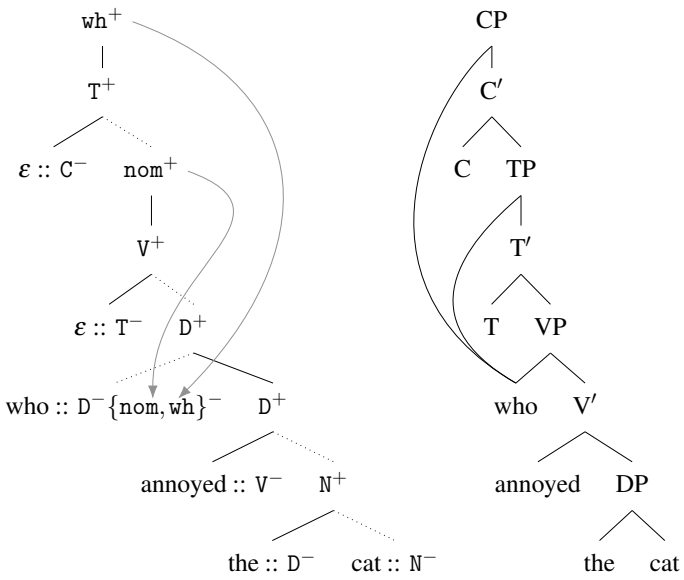
landing site that can check one of its licensee features.

The way movement is encoded in MG derivation trees is a bit unusual because no subtrees are ever moved into a different position — the actual displacement only happens during the construction of the derived structure, e.g. a bare phrase structure tree, a multi-dominance tree, a prosodic structure, or an LF-interpretation. As such, each Move step is merely a node in a derivation tree whose distribution is controlled by the presence of licensor and licensee features on LIs. Intuitively, the derivation trees are multi-dominance trees except that the dominance arcs introduced by Move are left implicit as they are easily inferred from the feature calculus. The trees in (21) each depict a derivation with `wh`-movement and `nom`-movement, with the corresponding multi-dominance tree on the right. In anticipation of the subsequent discussion, I already annotate all Merge and Move nodes with the positive features that host them. Following the discussion in Sec. 2.2, I also use arrows to indicate how each instance of Move is linked to a specific licensee feature — I will explain in a moment how these links are inferred.

(21) **Derivation tree with separate wh-movement and case movement**



(22) **Derivation tree with single phrase undergoing multiple movement steps**



While each derivation tree is almost identical to the corresponding multi-dominance tree, the dominance arcs in the latter do not quite mirror the feature checking relations in the former. Whenever an arrow ends in a licensee feature, the corresponding arc ends at the D-root of the LI that carries this feature (recall that the D-root of an LI is the highest node that it is connected to via a positive feature). That is to say, checking of a licensee feature on an LI l triggers movement of the whole phrase headed by l. In (21), the entire phrase *the cat* undergoes subject movement by virtue of being headed by *the*, which carries the relevant licensee feature $\texttt{nom}^-$.

Although readers familiar with Minimalist syntax won't have a hard time figuring out what features each Move node is hosted by, the issue merits discussion because it reveals profound parallels to Merge. In particular, the principles in (3) that connect Merge nodes to selector features and category features also work as intended for Move (provided that licensee features are unordered). The i-th positive feature of an LI is still connected to the i-th node above it, which is why we could use feature-annotated derivation trees in the examples above. And an interior node m has an outgoing arrow to some negative feature $\texttt{f}^-$ on LI l iff m is an $\texttt{f}$-occurrence of l. What more, Move is subject to all the constraints in (5)

that also apply to Merge: every Move node must be hosted by exactly one licensor feature (Single Head); each one of an LI's licensor features must host a Move node (Full Projection); and for every Move node *m* that is hosted by some $\texttt{f}^{+}$ there must be exactly one LI that *m* is an `f`-occurrence of (Match). Every aspect of MG feature checking that applies to Merge also holds for Move.

4.2 Move as an instance of TSL

The basics of Move in MGs have now been put in place. Over derivation trees, movement is a matter of feature checking, and as I explain next, the feature calculus for Move is an exact counterpart of Merge from the TSL perspective. The actual displacement of subtrees is a more complex affair, but at the end of this section I will briefly explain why it, too, is TSL.

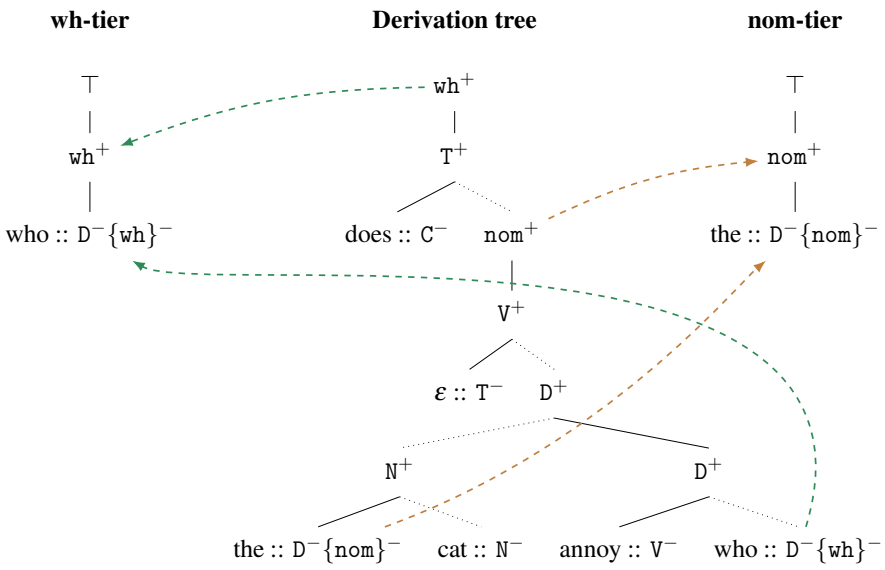
When viewed as a feature checking dependency, Move in our system has three essential properties: I) Move is triggered by (positive) licensor features and (negative) licensee features, and II) the licensee features of an LI are unordered, so that it always targets the closest possible Move nodes, and III) like Merge, Move is subject to the constraints Single Head (5a), Full Projection (5b), and Match (5c). These properties make it fairly easy to decompose Move into a number of constraints over derivation trees, which in turn are easily converted into local conditions on tiers that mirror those for Merge in Sec. 3.3. Among the three constraints Single Head, Full Projection, and Match, we already know that the first two are SL and can also be lexicalized via slices, so they need not be considered further (and I will only use feature-annotated derivation trees for the rest of the paper). Let us focus our attention, then, on how Match is enforced for Move.

Remember from Sec. 3.3 that the TSL view combines a projection function that truncates derivation trees into tree tiers with a licensing function that determines for every node on the tier what its strings of daughters may look like. For Merge, a tier is projected

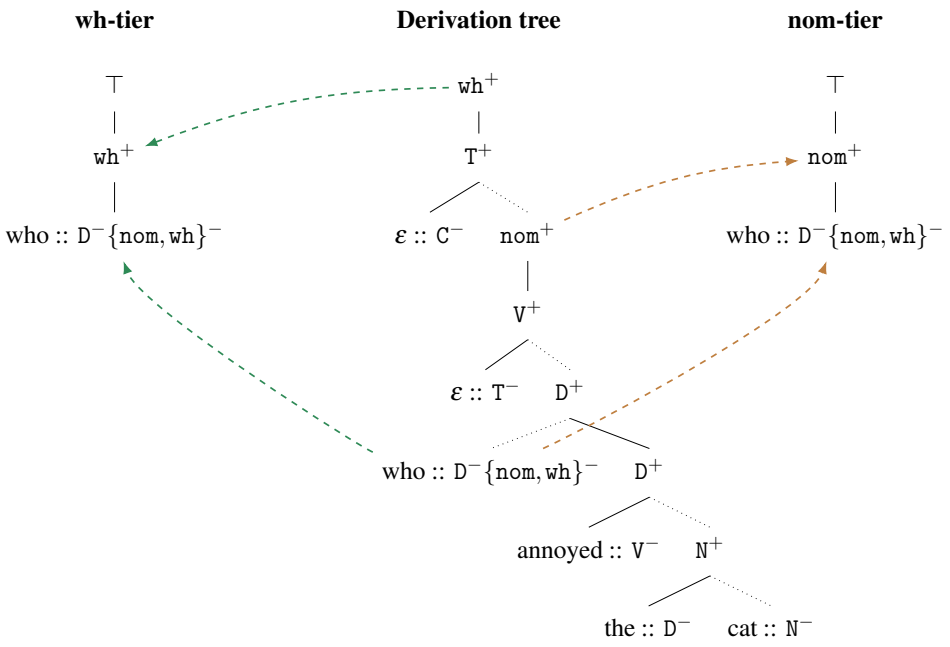
for every type of category, with each F-tier containing all Merge nodes that are hosted by $\texttt{F}^{+}$ and all LIs that carry $\texttt{F}^{-}$. On each tier, every Merge node must have exactly one LI among its daughters, and every LI must have a Merge node as its mother (which is enforced by disallowing lexical daughters for the tier root $\top$). The very same strategy works for Move: For each movement type f (e.g. nom, wh, ...) one projects a tier that contains I) all Move nodes that are hosted by some $\texttt{f}^{+}$, and II) all LIs with licensee feature $\texttt{f}^{-}$, and III) nothing else. Over these tiers, every Move node must have exactly one LI among its daughters, and every LI must have a Move node as its mother.

The examples below show how these conditions discriminate between well-formed and ill-formed derivation trees. The derivation tree in (23) is well-formed, and all its movement tiers contain a Move node with exactly one lexical daughter. The same is true for example (24) where a single mover targets multiple landing sites.

(23) **Well-formed derivation tree with tiers, multiple movers**

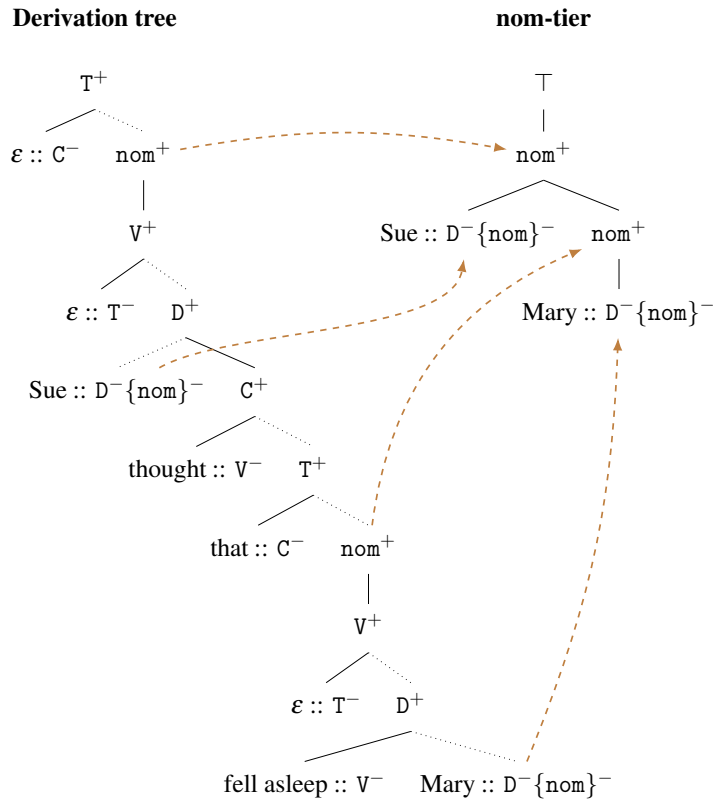


(24) **Well-formed derivation tree with tiers, single mover**



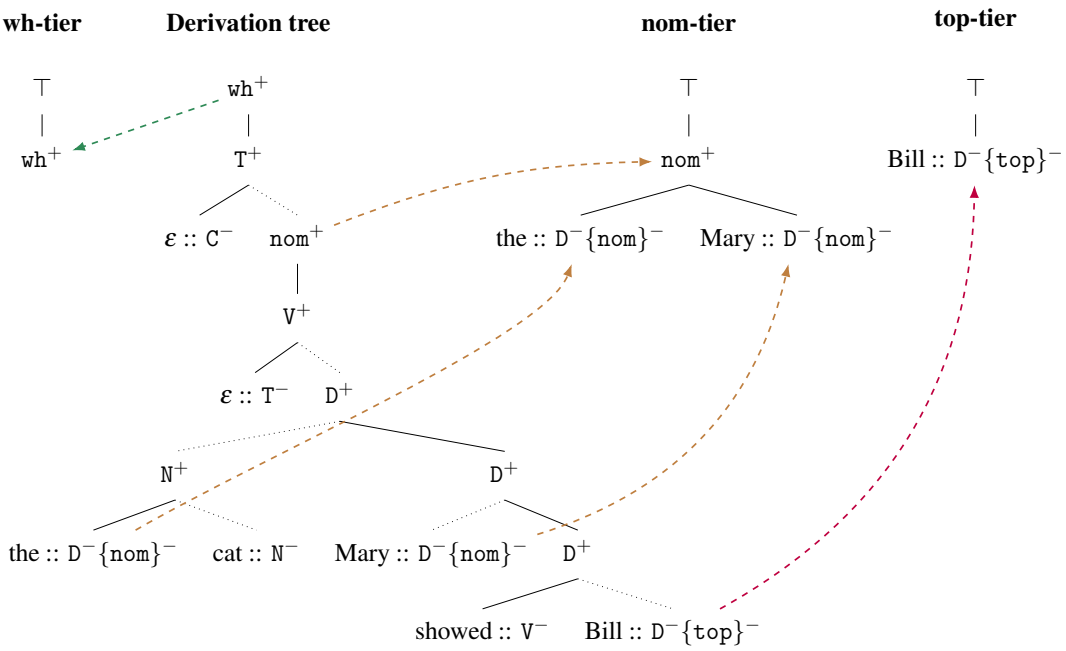
Example (25) shows that the move tiers of a well-formed derivation tree can also take on more complex shapes, but it still holds that every Move node has exactly one LI among its daughters and every LI has a Move node as its mother.

(25) **A well-formed derivation tree with a complex tier**



The ill-formed derivation tree in (26) illustrates how various violations are directly reflected on the corresponding movement tiers. Match is violated whenever a Move node has more than one lexical daughter, the lack of a mover is captured by the ban against Move nodes with no lexical daughters, and movers with no landing site are ruled out because the tier root $\top$ must have no lexical daughters.

(26) **Ill-formed derivation tree with tiers**



This shows that movement, when construed as a constraint on derivation trees, works exactly the same as Merge. The parallels become even more apparent if one compares the projection functions and licensing functions for Merge and Move.

(27) **TSL-comparison of Merge and Move (with feature annotations)**

	Merge	**Move**
Projection	LI with $\texttt{F}^-$	LI with $\texttt{f}^-$
	$\texttt{F}^+$	$\texttt{f}^+$
Licensing	$\texttt{F}^+ \mapsto \texttt{F}^+_{\geq 0}\ \texttt{F}^-\ \texttt{F}^+_{\geq 0}$	$\texttt{f}^+ \mapsto \texttt{f}^+_{\geq 0}\ \texttt{f}^-\ \texttt{f}^+_{\geq 0}$
	$\top \mapsto \texttt{F}^+_{\geq 0}$	$\top \mapsto \texttt{f}^+_{\geq 0}$
	$\top \mapsto \texttt{C}^+_{\geq 0}\ \texttt{C}^-\ \texttt{C}^+_{\geq 0}$ for C-tier	

The projection functions are exactly the same in all three cases, and only the licensing functions differ slightly because C-heads of matrix clauses do not need to be selected. We see, then, that the computational machinery that was needed for a more succinct description of Merge is exactly the same machinery that drives movement.

But there is one major difference: while Merge is still an SL-dependency and the step to TSL is motivated by succinctness and generality, Move is TSL but not SL. This is because there is no upper bound on the distance between a mover and the corresponding move node. There simply is no upper bound k such that a Move dependency never spans more than k levels in the derivation tree. In terms of shear complexity, then, Move outstrips Merge significantly. It is the need for a succinct specification that pushes us towards a TSL-view of Merge, which turns Move into a natural evolutionary outgrowth of Merge.

However, this finding only addresses movement in terms of the syntactic dependencies it induces, not in terms of the actual displacement of subtrees. But this, too, is a local process over tiers, although the relevant notion of locality is now one of structural rewriting. The mathematical concepts are a lot more involved in this case, so I will only present the general intuition. An output structure, e.g. a multi-dominance tree, is the result of carrying out the instructions specified in a derivation tree. This process is couched in mathematical terms as a function that maps derivation trees to output structures. The main challenge posed by movement is the association of a mover with its landing site(s). Intuitively, we want to connect each Move node m that is hosted by a licensor feature $\texttt{f}^{+}$ to the D-root of the LI l that m is an occurrence of. Thanks to the TSL-nature of Move, l is easily identified, though: it is a lexical daughter of m on the `f`-tier. Hence the actual displacement component of Move hinges on the ability to determine for any two nodes m and l whether l is a lexical tier daughter of m, which can already be done with the computational resources that are needed for TSL-Merge and TSL-Move. We see, then, that even the displacement aspect of Move emerges naturally from the TSL-view of Merge.

5 Conclusion

I have argued that a specific view of Merge and Move that is grounded in formal language theory, in particular subregular complexity, reveals surprising parallels between these two operations. Even though Merge is a strictly local (SL) dependency, it can be described in much more general, succinct, and less memory-intensive terms if one treats it as a restriction on mother-daughter configurations in tree tiers. But the resulting system is formally indistinguishable from Move, making the two essentially notational variants of each other. A possible evolutionary scenario, then, is the following: Merge started out as a maximally simple, strictly local operation that consumed a disproportionate amount of memory as grammars gradually increased in complexity. This created evolutionary pressure towards a more sophisticated, tier-based inference mechanism that consumes little cognitive resources no matter how large or complex the grammar. As soon as this mechanism was available, though, it was only natural to generalize Merge to Move. Since TSL also plays a central role in phonology and morphology, this story could be expanded into a general account that ties Merge and Move into developments beyond syntax.

This is, of course, just one of many conceivable scenarios. The argument is not that things must have happened this way, but merely that there is a plausible computational path from a local system of head-argument relations to a system with unbounded movement dependencies. Two properties set this path apart from alternative scenarios. First, SL is one of the weakest classes in formal language theory and has very few cognitive prerequisites (Rogers and Pullum 2011). Second, TSL is a minimal extension of SL, the only innovation being its ability to ignore symbols that are irrelevant for a given dependency. From a cognitive perspective, the step from SL to TSL, and by extension the step from Merge to Move, is not a big one. These findings are very much in line with Chomsky's original reduction of Move to Merge, but the argument from formal language theory does not hinge

on a specific set-theoretic definition of Merge or Move and instead derives directly from the computational nature of the syntactic dependencies that these operations establish.

Several interesting issues had to be omitted or left open. For instance, Graf (2018) argues that TSL-Merge is also a necessary evolutionary milestone on the way to adjunction, and the TSL-treatment of movement can also accommodate certain island phenomena such as the Adjunct Island Constraint, providing an evolutionary pathway towards restrictions on movement. At the same time, it is still unclear how well TSL fares with respect to head movement, sideward movement, Agree, and the many, many constraints that have been proposed in the Minimalist literature. In the spirit of this paper, though, I conjecture that subregular complexity provides a plausible evolutionary path from TSL-syntax to these extensions.

Acknowledgments

The work reported in this paper was supported by the National Science Foundation under Grant No. BCS-1845344. I thank the three reviewers for their very supportive comments, which led to several additions and changes in presentation.

References

Aksënova, Alëna, Thomas Graf, and Sedigheh Moradi. 2016. Morphotactics as tier-based strictly local dependencies. In *Proceedings of the 14th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 121–130. URL `https://www.aclweb.org/anthology/W/W16/W16-2019.pdf`.

Chomsky, Noam. 2004. Beyond explanatory adequacy. In *Structures and beyond: The*

cartography of syntactic structures volume 3, ed. Adriana Belletti, 104–131. Oxford: Oxford University Press.

De Santo, Aniello, and Thomas Graf. 2019. Structure sensitive tier projection: Applications and formal properties. In *Formal Grammar*, ed. Raffaella Bernardi, Gregory Kobele, and Sylvain Pogodalla, 35–50. Berlin, Heidelberg: Springer.

Graf, Thomas. 2012. Locality and the complexity of Minimalist derivation tree languages. In *Formal Grammar 2010/2011*, ed. Philippe de Groot and Mark-Jan Nederhof, volume 7395 of *Lecture Notes in Computer Science*, 208–227. Heidelberg: Springer. URL `http://dx.doi.org/10.1007/978-3-642-32024-8_14`.

Graf, Thomas. 2017. A computational guide to the dichotomy of features and constraints. *Glossa* 2:1–36. URL `https://dx.doi.org/10.5334/gjgl.212`.

Graf, Thomas. 2018. Why movement comes for free once you have adjunction. In *Proceedings of CLS 53*, ed. Daniel Edmiston, Marina Ermolaeva, Emre Hakgüder, Jackie Lai, Kathryn Montemurro, Brandon Rhodes, Amara Sankhagowit, and Miachel Tabatowski, 117–136.

Graf, Thomas. 2019. A subregular bound on the complexity of lexical quantifiers. In *Proceedings of the 22nd Amsterdam Colloquium*, ed. Julian J. Schlöder, Dean McHugh, and Floris Roelofsen, 455–464.

Graf, Thomas, Alëna Aksënova, and Aniello De Santo. 2016. A single movement normal form for Minimalist grammars. In *Formal Grammar: 20th and 21st International Conferences, FG 2015, Barcelona, Spain, August 2015, Revised Selected Papers. FG 2016, Bozen, Italy, August 2016*, ed. Annie Foret, Glyn Morrill, Reinhard Muskens, Rainer Osswald, and Sylvain Pogodalla, 200–215. Berlin, Heidelberg: Springer. URL `https://doi.org/10.1007/978-3-662-53042-9_12`.

Graf, Thomas, and Kalina Kostyszyn. 2021. Multiple wh-movement is not special: The subregular complexity of persistent features in Minimalist grammars. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2021*, 275–285.

Heinz, Jeffrey. 2018. The computational nature of phonological generalizations. In *Phonological typology*, ed. Larry Hyman and Frank Plank, Phonetics and Phonology, chapter 5, 126–195. Mouton De Gruyter.

Heinz, Jeffrey, Chetan Rawal, and Herbert G. Tanner. 2011. Tier-based strictly local constraints in phonology. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, 58–64. URL `http://www.aclweb.org/anthology/P11-2011`.

Kobele, Gregory M. 2011. Minimalist tree languages are closed under intersection with recognizable tree languages. In *LACL 2011*, ed. Sylvain Pogodalla and Jean-Philippe Prost, volume 6736 of *Lecture Notes in Artificial Intelligence*, 129–144. URL `https://doi.org/10.1007/978-3-642-22221-4_9`.

Richards, Norvin. 2016. *Contiguity theory*. Cambridge, MA: MIT Press.

Rogers, James, and Geoffrey K. Pullum. 2011. Aural pattern recognition experiments and the subregular hierarchy. *Journal of Logic, Language and Information* 20:329–342.

Stabler, Edward P. 1997. Derivational Minimalism. In *Logical aspects of computational linguistics*, ed. Christian Retoré, volume 1328 of *Lecture Notes in Computer Science*, 68–95. Berlin: Springer. URL `https://doi.org/10.1007/BFb0052152`.

Stabler, Edward P. 2011. Computational perspectives on Minimalism. In *Oxford handbook of linguistic Minimalism*, ed. Cedric Boeckx, 617–643. Oxford: Oxford University Press.