Interpreting Minimalist Grammars

Jake Mundo

April 16, 2018

1 Introduction

Minimalist grammars, popular in syntax, build sentences from lexical items using two basic operations: Merge and Move. Stabler (1997) gives a formal, mathematical account of minimalist grammars relying on these two operations in a way that allows such grammars to be treated computationally. With a formal approach to minimalist grammars, one can begin to ask several computational questions regarding such grammars, including, notably, the question of parsing: how can one determine the set of minimalist derivations that will lead to a particular sentence? Such a formal account has other benefits as well; for example, Lecomte (2008) uses a particular method of formalizing minimalist grammars, called minimalist categorial grammars, to establish strong connections between syntax and semantics. In general, exploring very formal encodings of minimalist grammars allows for a clear and extremely precise understanding of the behavior of the syntax, and the construction of careful and computationally relevant comparisons between syntax and semantics.

The main goal of this paper is to explain minimalist grammars such as those described in Lecomte (2008) and Fowlie and Koller (2017), to build a computational tool converting between minimalist derivational trees and more traditional minimalist syntax trees, and to consider possible theoretical implications of the computational perspective on the minimalist operations Merge and Move. First, we describe Koller and Kuhlmann’s (2011) formalism of interpreted regular tree grammars (Section 2) and how Fowlie and Koller (2017) encode a minimalist grammar as an interpreted regular tree grammar; furthermore, we discuss Fowlie and Koller’s derivational trees, which describe how a sentence is built up using Merge and Move (Section 3). We then present a different formalization of minimalism, which differs from previous approaches in that it treats the distinction between heads and phrases, as well as the head-complement-specifier relationship, as basic, rather than consequences of the structure of trees (Section 4). Finally, we implement code in Java which performs Merge and Move operations on these minimalist trees, and use that code to convert between
minimalist derivational trees, which describe how the tree is built up, and more traditional minimalist trees.

2 Interpreted Regular Tree Grammars

In this section, we describe the system of Interpreted Regular Tree Grammars (IRTGs) developed in Koller & Kuhlmann (2011). An IRTG consists of a language of derivation trees described using a Regular Tree Grammar (RTG), together with an interpretation of these derivation trees into an algebra. By varying the algebra and the interpretation of the RTG into the algebra, we vary the formalism into which the IRTG is interpreted; for example, there are distinct algebras for Tree Adjoining Grammar (TAG) and Featural Tree Adjoining Grammar (FTAG), and our goal will be to build the algebra and the interpretation for FTG. The key to the flexibility of IRTG is that it can be adapted to different systems simply by changing the algebra and the interpretation, while maintaining a similar underlying language of derivation trees.

This section proceeds in two parts: first, we define precisely what we mean by an RTG and a language of derivation trees; then, we define the interpretation of an RTG into an algebra.

2.1 Regular Tree Grammars

Intuitively, an RTG is a structure by which we can take trees and compose them into larger trees. It is analogous to context-free phrase structure grammars in syntax. Essentially, we have a particular tree we call $S$, which represents a sentence, and rules by which we can break down $S$ into symbols. Eventually, we reach terminal symbols that we do not break down further. There are many ways we could break down $S$, and we call the set of ways to break down $S$ the language of the RTG. In other words, the language of the RTG is the set of syntax trees we can build from the context-free phrase structure grammar rules which define the RTG.

To define an RTG more formally, we will need the following terminology. All definitions in this section come from Koller and Kuhlmann (2001).

Definition. A signature is a finite set $\Sigma$ of symbols $f$, each assigned a non-negative integer as its rank. This rank corresponds to the number of children a node labeled $f$ has in a tree.

Definition. A finite constructor tree over $\Sigma$ is a tree whose nodes are labeled with symbols $f \in \Sigma$, where a node labeled $f$ has exactly the rank of $f$ children. We denote by $T_\Sigma$ the set of all finite constructor trees over $\Sigma$. 
To describe the tree with root labeled $f$ and subtrees $t_1, \ldots, t_n$, we will write $f(t_1, \ldots, t_n)$. This notation evokes the intuition that the node $f$ takes in its children as inputs, like a function, and outputs the entire tree.

Now, we define what it means to be an RTG. Intuitively, an RTG is a set of production rules which allow us to change a nonterminal symbol into a set of new symbols. For example, an RTG may include rules like $S \to NP \ VP$ or $VP \to V \ NP$, or even $V \to eat$. To define an RTG more formally, we need to formally distinguish between the nonterminal symbols (such as NP and VP), which can expand into more, and the terminal symbols (generally lexical items), which cannot.

**Definition.** A *regular tree grammar* is an ordered 4-tuple $G = (N, \Sigma, P, S)$ where $N$ is a signature of nonterminal symbols, $\Sigma$ is a signature of terminal symbols, $P$ is a set of productions, and $S \in N$ is a distinguished start symbol. A *production* is a rule $B \to t$, where $B$ is a nonterminal symbol and $t$ is a tree over $\Sigma \cup N$, which replaces $B$ with the tree $t$.

Note that the production rule is not quite of the form $S \to NP \ VP$, but rather $S$ goes to the tree with a root $f$ and two leaves NP and VP, which we can write $f(NP, VP)$.

The last definition in this section concerns all the different trees which can be built up using the RTG.

**Definition.** The *regular language* $L(G)$ generated by $G$ is the set of all trees $t$ over $\Sigma$ that can be derived from $S$ using production rules in $P$.

We can think of the regular language as the set of all completed syntax trees describing a sentence. The condition that the trees be over $\Sigma$ ensures that only terminal symbols are used.

As an example, consider the following simple (and linguistically interpretable) RTG. Our nonterminal symbols ($N$) are $S$, $NP$, $VP$, $V$, $N$, and $D$. Our terminal symbols ($\Sigma$) are $\alpha_S$, $\alpha_{NP}$, $\alpha_{VP}$, $\alpha_V$, $\alpha_N$, $\alpha_D$, $the$, $dog$, $cat$, and $sees$. (We will see the purpose of these symbols soon.) We have the following productions in $P$:

- $S \to \alpha_S(NP, VP)$
- $NP \to \alpha_{NP}(D, N)$
- $VP \to \alpha_{VP}(V, NP)$
- $D \to \alpha_D(\text{the})$
- $N \to \alpha_N(\text{dog})$
- $N \to \alpha_N(\text{cat})$
- $V \to \alpha_V(\text{sees})$
The special start symbol is, unsurprisingly, the sentence symbol $S$. This RTG encodes a very small grammar fragment. The nonterminal symbols are syntactic categories, and the terminal symbols are of two sorts: lexical items, which end up the leaves of the tree, and the $\alpha$s, which are the nonterminals with which we replace $S$, NP, and so on after they are expanded.

Let us consider an example of (part of a) derivation:

<table>
<thead>
<tr>
<th>Step of Derivation</th>
<th>Production Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>start node</td>
</tr>
<tr>
<td>$\alpha_S(NP, VP)$</td>
<td>$S \rightarrow \alpha_S(NP, VP)$</td>
</tr>
<tr>
<td>$\alpha_S(\alpha_{NP}(D, N), VP)$</td>
<td>$NP \rightarrow \alpha_{NP}(D, N)$</td>
</tr>
<tr>
<td>$\alpha_S(\alpha_{NP}(D, N), \alpha_{VP}(V, NP))$</td>
<td>$VP \rightarrow \alpha_{VP}(V, NP)$</td>
</tr>
<tr>
<td>$\alpha_S(\alpha_{NP}(D, N), \alpha_{VP}(V, \alpha_{NP}(D, N)))$</td>
<td>$NP \rightarrow \alpha_{NP}(D, N)$</td>
</tr>
</tbody>
</table>

At this point, we only need to apply the rules that take $D$, $N$, $V$ to lexical items to get a structure in the language defined by the RTG. We give an example of one such structure, written out fully as a tree:

```
\alpha_S
  \alpha_{NP}  \alpha_{VP}
    \alpha_D \alpha_N  \alpha_V \alpha_{NP}
      the cat sees
        \alpha_D \alpha_N
          the dog
```

The language of this RTG includes this tree and three others, corresponding to the different possibilities of $cat$ and $dog$. Of course, any RTG which would be used in practice would be much larger than this, but this RTG illustrates the general concept.

We readily recognize the tree above as a tree for the sentence *The cat sees the dog*. However, in the RTG, the tree is simply a tree, and does not correspond to any sentence or similar structure. In order to make the tree correspond to a sentence, we have to interpret it. This is the basic idea behind the IRTG: we assign to each tree in the RTG’s language an interpretation in some algebra. In the algebra of linear strings of words, the tree above would be interpreted as *The cat sees the dog*. We will discuss the process of interpretation in more detail in the next section.
It is important to note that the nodes in an RTG, even an RTG corresponding to a sentence, may not so straightforwardly correspond to syntactic types. The minimalist RTGs we will later see will have nonterminal nodes corresponding to the operations Merge and Move and terminal nodes corresponding to lexical items; while this structure bears similarity to the simpler RTGs seen in this section, it should be noted that they are not the same. We do not work any further with the RTG described in this section, and it was introduced purely for simplicity describing how an RTG works.

2.2 Interpretation of an RTG

In this section, we discuss the formal construction of an Interpreted Regular Tree Grammar (IRTG). All definitions come from Koller (2015), and are essentially the same as those in Koller and Kuhlmann (2011). We first will discuss intuition on the idea of interpreting an RTG, and then give the formal definition.

Intuitively, a tree in the language defined by an RTG can be interpreted as another structure, which we call an algebra. One simple algebra is the algebra of strings of English letters, which includes elements like the dog, the cat, the cat sees, aaaaaaa, and the cat sees the dog. The tree in the previous section should be interpreted as the cat sees the dog. In this case, it is straightforward to compute the interpretation directly from the tree, but one can imagine that it could be useful to build up the interpretation in the string algebra parallel to building up the tree.

The string algebra is not the only algebra into which we can interpret trees given by an RTG. We can also interpret these trees into sophisticated encodings of the syntax, such as trees of Featural Tree Adjoining Grammar (FTAG) or featural structures of Lexical Functional Grammar (LFG). In this paper, we focus on one broad type of encoding, minimalist trees. The essential step in describing the interpretation of an RTG is defining how we build up interpretations parallel to how we build up the RTG tree. We now proceed to define this in detail.

Earlier, we mentioned that the notation $f(t_1, \cdots, t_n)$ evokes the intuition of a function. In fact, when we perform the interpretation of an RTG, we want to interpret a symbol $f$ of rank $n$ as a function which take in $n$ inputs and outputs a single output, corresponding to the behavior of taking in its $n$ distinct children (all trees of their own) and making a single tree out of them, with its own node as the root. To this end, we have the following definition:

**Definition.** A $\Sigma$-algebra $A$ consists of a non-empty set $A$ called the domain, and for each symbol $f \in \Sigma$, a function $f^A : A^n \rightarrow A$ which takes in $n$ elements of $A$ and outputs a single element of $A$. 


As an example, we can first consider the \( \Sigma \)-algebra of strings. Each of the symbols in \( \Sigma \) – that is, the lexical items and the \( \alpha \)s, in the case of the above example – are given an interpretation. For lexical items, that interpretation is just the word itself, so \( cat^A \) is the string \( cat \). (This is a zero-place function, as \( cat \) has rank \( 0 \), and hence is just a constant.)

The interpretation of the \( \alpha \)s is fairly straightforward in this small example: all of them interpret to functions which simply concatenate the two strings they are given. The interpretation functions are not, in general, this straightforward, but with this simple case, concatenation is sufficient.

In general, we define the interpretation of an entire tree recursively with the following formula:

\[
[[f(t_1, \cdots, t_n)]]_A = f^A([[t_1]]_A, \cdots, [[t_n]]_A).
\]

In other words, we straightforwardly apply each node as a function to its children to build up the interpretation of the entire tree. In the case of the string algebra, this recursive definition successfully gives the interpretation of the tree in the previous section: \textit{The cat sees the dog}.

3 Minimalist Grammars

In this section, we will discuss minimalist grammars, and how they relate to the interpreted regular tree grammars discussed above. In particular, we focus on work by Fowlie and Koller (2017), which encodes a minimalist grammar as an interpreted regular tree grammar. The descriptions of minimalist grammars in this section all come from Fowlie and Koller (2017).

3.1 Minimalist Grammars, Informally

Informally, in a minimalist grammar, we build up representations of sentences from basic lexical items using the operations Merge and Move. The operation Merge takes two constituents and combines them, while the operation Move takes an element within the structure and moves it to a higher position in the structure. Features on these lexical items motivate the operations.

Let us give an example. The following example comes from Fowlie and Koller (2017), although it is given a more traditional and less formal presentation here. Consider the English sentence:

(1) Who laughed?
A (somewhat simplified\(^1\)) standard minimalist syntax tree for this sentence is the following:

![Syntax Tree](attachment:image.png)

The lexical item *who* is base-generated as the complement of the verb *laughed*, and then moves up to the specifier of tense T, then to the specifier of the question complementizer C. Some feature on each of T and C prompts this movement.

This structure is built up by a sequence of Merge and Move operations on the lexical items. For example, *laughed* and *who* merge to the VP *laughed* *who*. This Merge is possible because *laughed* selects a D (or DP) complement, such as *who*; this selection relationship is encoded in features on V. Similarly, *who* moves up in the structure because it has a feature that is attracted by features on T and C.

In the next section, we will state precisely, following Fowlie and Kohler (2017), how such a minimalist grammar functions.

### 3.2 Minimalist Grammars, Formally

In the example (1) in the previous section, we saw two main types of features: those involved in selection (Merge) and those involved in attraction (Move). Following Stabler and Keenan (2003), we can distinguish, formally, two finite disjoint sets of features: selectional features, which drive Merge, and licensing features, which drive move. Selectional features might be syntactic categories like V or D, while licensing features might be properties like the question feature wh.

\(^1\)We simply write D for the category of *who* rather than DP, for the sake of simplicity.
Each type of features can have positive or negative polarity. For selectional features, for example, the verb laughed has a feature which selects for something with the feature D. Following Fowlie and Koller (2017), we denote this feature =D, so =D and D are opposite polarity selectional features. Similarly, licensing features have positive and negative polarities; the complementizer has the feature +wh, which attracts a wh-word, that is, a word like who with the feature -wh. Attraction is governed by the Shortest Move Constraint (SMC) (Fowlie and Koller 2017), which enforces that a feature +f can only attract the nearest item with the feature -f, and consequently that only one item with feature -f can be present in the syntax in a not-yet-attracted state at a time.

In order to formally define minimalist grammar, we need to define what it is to be a minimalist structure. An intuitive method would be to define a minimalist structure to be a minimalist tree; we will return to this idea later. For the moment, we follow Fowlie and Koller (2017) and define a minimalist structure in the following way: Let a string with features be a string (of words) together with a set of features; for example, the word who together with the features D, -nom, -wh is a string with features. We might want to say that a minimalist structure is then just such a string with features, but we also need a way to encode items within the string with features which might get moved out; to do so, we add a storage, a function from licensing features to other strings with features. For example, we might have in the storage that the licensing feature nom will attract the string with features who. Because of the SMC, there will only be one string with features waiting to be attracted by a particular feature at a time, so this function representation works.

With the notion of storage, Fowlie and Koller (2017) define an expression to be a string with features together with a storage. These are the objects of the minimalist grammar, and they are then able to define the merge and move operations; we do not go into the formal details of those definitions here, but describe them informally.

First, the Merge operation takes in two expressions a with a feature =X and b with a feature X and merges them into a new expression c. The feature X is responsible for driving the Merge operation. If b does not have any licensing features, Merge simply concatenates the strings, cancels a feature =X on a and X on b (this feature prompted the merge operation), and combines the storages. If b does have licensing features, then b is added to storage under those features, so that b can be attracted by them.

The Move operation takes a single expression a and outputs an expression c, the result of moving something in a with feature -f up to specifier position of something with feature +f. The Move operation cancels

---

2Fowlie and Koller do not give a name to these objects, but we introduce one for convenience.
those +f and -f features (the -f feature is canceled by removal from storage) and moves the string up. The only remaining operation is to update storage; for example, if storage previously said that the feature nom will attract who, which has the feature -wh, it will be updated to say that now the feature wh will attract who, which has no remaining features. In this way who can be attracted multiple times.

This minimalist system is useful and allows for a formal discussion of minimalist grammars, as it precisely defines what an element of a minimalist grammar is. However, this notion of expressions is not necessarily the most straightforward way to think of a minimalist object; indeed, traditionally, we tend to think of minimalist objects as trees. Therefore, purely for illustrative purposes, it is useful to see how a minimalist grammar will work without the notion of storage, instead relying straightforwardly on minimalist syntax trees as representations. This will be the main work of the next section.

3.3 A Minimalist Grammar of Trees

In this section, we define a minimalist grammar on trees. This objective is mostly valuable for illustrating a formal minimalist grammar in a more familiar and intuitive setting, as well as giving a more complete representation of the minimalist syntax of a sentence in a computationally viable way. The definitions are based on Fowlie and Koller’s (2017) storage-based minimalist grammar, modified to replace storage with a powerful tree representation.

Here, an object in the grammar is a binary tree. Each nonterminal node of the tree has a category, which will be either X’, or XP for some lexical category X. They also have associated strings, as well as features such as D or -nom, and hence can be selected or attracted. Terminal nodes have a category X, an associated string, and features as well; however, the features of a terminal node can also be of the sort that attract or select, such as +nom and =D, as well as the sort that can be attracted or selected. All these features can be marked as satisfied or not, depending on whether they have been canceled with the appropriate feature of opposite polarity. For example, when an item with a D feature merges with an item with a =D feature, both are satisfied.

Furthermore, a tree has movement arrows, which are pairs of nodes one of which moves to the other. A tr equipped with its movement arrows is an object of the grammar.

Merge and Move, then, are now operations on trees. Merge takes two trees of the form X and YP and merges them into a structure
(or the same with X and YP reversed\(^3\)). The features on the XP node will be precisely the features of the form \(=A\) and \(-f\) on the X node.

Move takes a single tree of the form

\[
\begin{array}{c}
XP \\
\uparrow \\
X \ YP
\end{array}
\]

where X has an unsatisfied feature \(+f\) and there is some ZP in the YP with an unsatisfied feature \(-f\). It then produces

\[
\begin{array}{c}
XP \\
\uparrow \\
ZP \ X' \\
\downarrow \\
X \ YP
\end{array}
\]

In the above tree, ZP ends up as the specifier of X, and is moved or copied from somewhere within YP.

As an operation on trees, Move seems rather complicated, as it replaces the left child of XP with ZP, replaces the right child with the somewhat mysterious \(X'\) node, and moves the former children of XP to be children of \(X'\). It is worth noting that the operation is less complicated if we view an implicit \(X'\) present in the original tree, so it is actually

\[
\begin{array}{c}
XP \\
\uparrow \\
X' \\
\downarrow \\
X \ YP
\end{array}
\]

However, it is inelegant to include this apparently unnecessary \(X'\) node, so we do not draw our trees with it. Move is also much less complicated if we view it as simply adding a specifier, and avoid considering the tree structure. This is the approach we adopt in Section 4.

It is also worth noting that when the ZP moves out of YP, we do not delete it from inside the YP; in other words, we adopt a copy approach to movement. This is purely a matter of convenience.

\[\text{--- in this paper, we assume a head-specifier order. It can be assumed that the order is determined either by a language parameter or by some feature on the head.} \]
These definitions of Merge and Move describe the minimalist grammar in a formal way. The motivation for the definitions in the previous section are now somewhat more evident; they simply allow all the needed information to be stored on a single node, including the information about what lower nodes can be moved up. In other words, Move can be a one-place operation, which takes a tree and finds the right constituent to move from inside it.

This minimalist grammar of trees makes clear a systematic distinction between lexical items like X and complete phrases like YP, which behave very differently with regard to Merge and Move. The minimalist grammar in this section does not explicitly acknowledge that distinction, but in Section 4, we build a minimalist grammar that specifically requires that Merge take a head and a phrase as an argument, and that Move take a phrase as an argument. This choice leads the grammar to better reflect the syntactic distinction between heads and phrases.

### 3.4 Minimalist Derivation Trees

We now return to the RTGs and IRTGs from earlier in the paper. Fowlie and Koller (2017) construct an interpretation from RTGs into a minimalist grammar. Their RTGs have as nonterminal nodes Merge and Move, and as terminal nodes lexical items written as expressions. These RTGs carefully encode the different versions of Merge and Move for slightly different inputs, but the authors give simplified derivation trees which are similar. For example, recall (1) above, restated here as (2):

\[(2) \text{ Who laughed?}\]

If we assume the minimalist structure given above, then a derivation tree (similar to an RTG with Merge and Move nodes) of this sentence is

```
move
merge
(ε, =V+whC)
move
merge
(ε, =V+nomT)
merge
(laughed,=DV) (who, D-nom-wh)
```
This tree lists all the featural information relevant to the sentence, which was absent in the previous minimalist tree. In particular, it identifies the categories complementizer (C) and tense (T) of certain lexical items, as well as all of their features, such as the verb-selecting feature \( =V \) or the nominative-attracting feature \( +\text{nom} \). The symbol \( \epsilon \) simply denotes that there is no string corresponding to that lexical item, that is, it is silent.

The notation for the expressions simply encodes both their string and their set of features; for example, \( (\text{who}, \text{D-nom-wh}) \) describes the word \textit{who} with features D, -nom, and -wh. This derivation tree encodes precisely how we build up the minimalist tree, and indeed from it we could build up the minimalist syntax tree given above. Doing so is essentially interpreting (in just the sense of the IRTG) this derivation tree into an object of the minimalist grammar.

4 A Minimalist Grammar Based on the Head-Complement-Specifier Relationship

In the previous section, we discussed Fowlie and Koller’s (2017) approach to Minimalist grammars, which relies solely on the operations Merge and Move. In that approach, Merge is always used to add something new, and Move is never used to add something new. In both the original storage-based grammar and the grammar of trees, this is the case.

However, sometimes, it is helpful to enrich Minimalist grammars with more tools, such as head movement (Amblard, Lecomte, & Retoré 2010). Although head movement does not actually increase the expressive capacity for the grammar (Amblard, Lecomte, & Retoré 2010), it is linguistically valuable.

Motivated by an interest in a closer connection to the linguistics, in this section, we provide a new and slightly different minimalist grammar than that of Fowlie and Koller (2017), much more like that of Amblard, Lecomte, and Retoré (2010), although not the same. We introduce a systematic distinction between heads and phrases, a distinction standard in syntax but not present in any of the formal systems so far presented. This distinction will allow us to see head movement as a special sort of Merge (not Move), following Amblard, Lecomte, and Retoré, and furthermore will allow us to see the addition of specifiers as an operation more akin to Move than to Merge. The critical novel aspect of the grammar introduced here is that it takes the head-complement-specifier relationship as the basic structure of the grammar, rather than a tree structure (although we still draw trees), which allows this system to interface well with many Minimalist analyses.

At the end of this section, we will outline how to convert this sort of Minimalist grammar into an IRTG,
very similarly to what Fowlie and Koller (2017) do for their Minimalist grammar. Although the expressive power of this Minimalist grammar is not greater than theirs, it may be a valuable way to encode a grammar from a linguistic perspective, where a distinction between heads and phrases may be important.

4.1 Formalism for a New Minimalist Grammar

Once again, we build a minimalist grammar based on trees, and therefore not strictly relying on the idea of storage. However, we do use the idea of storage in a concrete implementation of the grammar in Java for computational ease.

Formally, we say a **head node** is a string together with a set of features, much like above; these features will be licensing features and selecting features, as well as a new type of feature, called a head-selecting feature, which will drive head-movement. One such feature might be a feature which attracts the head V. For our grammar, we will assume that a head can only attract the head of its complement, so we can treat this feature as boolean: either a head does attract its complement’s head, or it does not. An example could be a verb moving from head to head from the base V to various Voice and v heads up the structure.

A **phrasal node** XP is an ordered triple (X,YP,ZP), where X is a head node and YP and ZP are either empty or phrasal nodes. YP is called the **complement** of the phrasal node, and ZP is called the **specifier**.

Note that this is the first time the notions of complements and specifiers have appeared in any of the formalized minimalist systems we have looked at; in this way this system is more directly linked to theoretical minimalism, at least variants that use complements and specifiers. We will not use adjuncts in this system, instead assuming that everything is introduced in specifier positions.

Traditionally (where there is a specifier-head-complement word order), the phrasal node XP is drawn as something like

```
    XP
   /\    ;
  ZP   X'
    /\    
   X   YP
```

however, nowhere in the formal system we develop here is there any particular claim as to the relative position in such a tree of the complement or specifier. Trees are not formal objects in this grammar, and we draw them this way simply because it is convenient; formal objects in this grammar are phrases, which consist of a head, possibly a complement, and possibly a specifier. The grammars we have seen before have the children of a node as fundamental concepts, and we can define complement or specifier structurally from
them; in contrast, this grammar has complement and specifier as fundamental concepts, and the trees we build simply describe what is the complement and what is the specifier of what head in the usual way.

Movement is always from a phrasal node to a phrasal node. We assume the destination of movement is always in a specifier position.\(^4\)

The operation Merge is straightforward, and in this grammar is always a function of a head node and a phrasal node. Merge takes a head node X and a phrasal node YP and merges them into a new phrase XP with head X, complement YP, and no specifier. Unlike the previous systems, we never use Merge to add a Specifier, as that is a structurally different operation.

Move is the primary operation for introducing specifiers. Move takes an XP as its sole argument. It looks at the features on the head of the XP, and finds an attractor feature +f; it then finds the feature -f on some ZP within the YP. That ZP is then made to be the specifier of the XP.

We have not yet explained how to introduce new arguments. Formerly, when we wanted to introduce, for example, the subject of the sentence in the specifier of some phrasal node like vP or TP, we did so via Merge. However, for structural simplicity, we are now requiring that Merge only combine a head node with a phrasal node. The operation we have seen used to add specifiers is Move. Therefore, to introduce a new specifier, we actually use a variant of Move, which takes two arguments XP and ZP and makes ZP the specifier of XP. This second argument ZP was not part of the sentence before. Like ordinary Move, this two-argument Move (which is not anything we would traditionally call syntactic movement) requires that the ZP has a +f feature to match the -f feature on the head X of the XP.

We choose to call this operation Move as well, since it is really structurally identical to Move. Indeed, we could even consider them both a single operation with two arguments, but where the second argument (ZP) is allowed to already be a constituent of the first argument (XP), or not.

Our new minimalist grammar also includes head movement, that is, movement from a head to a higher head. As an example, suppose we claim there is V to T movement in a particular language, like French; then we would have the T node, which selects a VP, have a feature which prompts movement of the V head of the VP. We call this process movement, but structurally, it has little in common with the Move of the previous section; it does not move phrases, and its destination is not a specifier. In fact, this operation has much more in common with Merge, and indeed can be said to happen alongside Merge: when X and YP merge, Y moves up to X. Therefore, we choose to consider head movement a special case of Merge. This is

\(^4\)Note that in this system we do not assume the starting point of movement is also always a specifier position; however, this would not be an unreasonable assumption, and would only require that most items be introduced in specifier position. Indeed, this might make for a cleaner formalism.
the same approach as that taken by Amblard, Lecomte, and Retoré (2010).

Sometimes a phrase does not have a complement or a specifier, only a head. Since Merge is our operation for taking a head and creating a phrase out of it, we have a special version of Merge that takes only a head and outputs a phrase with that head. It may be misleading to call this operation Merge, since it only takes one argument; however, the motivation for this name is clearer if one sees it as Merge of a head and a null complement.

To summarize, in the previous minimalist grammars we have seen, there are two operations Merge and Move divided in the following way:

1. Merge takes two arguments and outputs one.
2. Move takes one argument and outputs one.

These operations both act on trees, so their arguments and outputs are trees. Our minimalist grammar also has two operations Merge and Move, but they are divided differently:

1. Merge takes a head (and possibly a phrase) and outputs a phrase. (Alternatively, Merge introduces complements.)
2. Move takes a phrase (and possibly another phrase) and outputs a phrase. (Alternatively, Move introduces specifiers.)

In particular, in our system, head movement is Merge, introduction of a specifier is Move, and projection of a head into a phrase is Merge. Furthermore, the arguments and outputs of our operations are heads and phrases, where phrases are a head grouped with a specifier phrase and complement phrase; while we can draw these structures as trees, we do not treat them formally as trees. All these aspects differ from other computational implementations of minimalism, and explicitly reflect the head vs. phrase distinction and head-complement-specifier relationship generally used in minimalist syntax.

4.2 A Computational Implementation

We have written Java code which serves several functions. First, and most importantly, it formally encodes the minimalist grammar described here, and can perform the operations of Merge and Move. Second, it can interpret derivation trees, in the sense of the interpretation function of an IRTG; in other words, given a derivation tree, it can produce the corresponding minimalist tree. It also performs the reverse operation,

---

5All code written for this thesis is available at https://github.com/jdmundo/ling-thesis.
taking a minimalist tree and outputting its corresponding derivation tree. For ease of use, it can also print the derivation trees and minimalist trees in LaTeX code for use with the tikz-qtree package.

At the moment, the implementation is limited by specifier-head-complement word order. It could, however, be converted to handle other word orders. Because the code does not parse into strings, but rather converts between trees, this word order is really only relevant in the way the trees are printed, and is not a deep issue with the program.

We store minimalist syntax trees in terms of their top phrasal node. A phrasal node is defined in terms of its head node and two other phrasal nodes, that is, its complement and specifier. Similarly, we store derivational trees in terms of derivation nodes, which have either a nonterminal node like merge or move, or a terminal node which contains lexical information in the form of a string and a feature set.

To convert derivational trees into minimalist trees, we start on the top node and recurse to build the trees on the child nodes. A merge node merges the structure on the left as the head and the structure on the right as the specifier. It then updates the storage to correctly indicate what items are ready to be moved. If a head-attracting feature is present on the head, it moves the head as well.

A Move node takes the given structure, reaches into storage, and pulls out the unique structure attracted by the head\textsuperscript{6}; alternatively, if given two arguments, the move node makes the first the specifier of the second. This is to reflect the fact that we generally use Move to add specifiers, even if they do not come from deeper in the structure. In this way the term Move is not necessarily appropriate, as there is no real movement happening, but the operation is formally similar, so we call it Move.

Similarly, to convert minimalist trees into derivation trees, we start with the top node; if it has a specifier, we add a move node to introduce it, and if it does not but does have a complement, then we add a merge node to introduce its complement. In this way we can build up the entire derivation tree. Note that we need to make use of feature information on the nodes which is stored in the implementation, but not printed in the drawings of the trees shown here.

As an example, the derivation tree above, from Fowlie and Koller (2017), can be easily modified to a derivation tree in this minimalist grammar; we only need to add one single-argument Merge step to change the D who into a DP who. This gives the derivation tree:

\textsuperscript{6}That only one structure is attracted by the head is a consequence of the Shortest Move Constraint (Stabler 1997), which requires that two features that can be attracted are active in the storage at the same time only if they are distinct.
Our code can convert between the trees of these types, as well as print usable LaTeX code for the trees. The LaTeX giving the trees above was generated directly by the program, and the program converts correctly between these trees.

It is worth noting that the Java implementation of minimalist grammar is a first step toward encoding this new minimalist grammar as an IRTG. This would complement Fowlie and Koller (2017)'s encoding of their minimalist grammar as an IRTG, and would allow flexibility of what sort of minimalist grammar is used in a computational system. The ability to translate from derivational trees to minimalist trees also
complements the parsing task which could be facilitated by an encoding as an IRTG, as it allows the resulting parses to be immediately translated into actual minimalist structures.

4.3 Argument Introduction

The essential goal of the minimalist grammar described above is to maximize simplicity on the computational side, while still leaving the possibility for complex linguistic processes to be encoded as minimalist grammars. This section demonstrates how we might encode a particular complex structure in a grammar of this sort.

With the goal of simplicity, we have imposed several very strong restrictions, including:

1. Each head can select at most one phrase as its complement.
2. Each head can attract at most one phrase as its specifier.
3. Each new element is introduced in the specifier position.
4. Head movement only occurs alongside Merge.

These conditions are rigid, but we demonstrate that even sophisticated structures can be expressed using them, and they are computationally elegant in their simplicity, compared to approaches that allow, for example, Merging of two phrases or of an XP and a Y'. It is important to note that these restrictions make it impossible to encode certain apparently simple structures into this minimalist grammar. For example, a causative transitive verb would seem to take three arguments: a causer, an external argument, and a direct object. How should those arguments be introduced? In this section, the primary goal is to describe how argument introduction works with the minimalist grammar described above. The analyses in this section are not unusual in the context of the Minimalist program, but are presented specifically with regard to the Minimalist grammar above.

As a particular example, consider the following sentence in the Bantu language Kikuyu:

(3) Kuria niambithia Kamau hema.

Kuria pitched -CAUS -CAUS -FV Kamau tent
‘Kuria has made Kamau pitch a tent.’

In order to describe this sentence in a minimalist grammar of the sort we have just described, we can use a tree of a form something like the following:

---

7The example comes from Waweru (2011). The FV GLOSS MEANS FINAL VOWEL AND IS UNUSUAL TO KIKUYU; SOME FINAL VOWEL APPEARS ON EVERY VERB, BUT THEY ARE NOT RELEVANT HERE.
We give content words in English in the tree for clarity. The specific choices of the names Voice and v for the heads are not relevant to this particular analysis; the choice to separate the Voice and causative heads follows Harley (2013) but has no significant bearing on the computational implementation. Similarly, the names of the features we choose in the following derivational tree are not important; they simply need to perform the correct attractions and selections.

Notice that in the above tree, each new argument is introduced in its own specifier by argument-introducing heads. We give each of these heads some feature attracting a DP specifier.

The derivation tree, then, for the above minimalist tree would be something like the following (only those features relevant to this part of the derivation are shown; there will be more features to account for movement up to T, but this behavior is similar to the previous example, so is omitted here):

```
move
\langle Kuria,D-causer\rangle
```

merge

```
\langle \epsilon,=v_{CAUS}Voice+causer\rangle
```

merge

```
\langle -ithi,=Voice_{CAUS} \rangle A
```

where the tree denoted A is
(The two trees are separated not for any systematic reason but simply because they are too large to be
drawn together; they really should be considered one tree.)

In this tree, the features we call causer, ext, and obj are simply used to indicate the roles played by the
nominals in the phrase. We take no stance as to what they are – it may be useful to think of them as theta
roles, for example. However, for the system to successfully allow the heads to introduce these nominal items,
there must be some feature in them which the heads select for.

This example illuminates how, in this minimalist approach, all of the content and featural information
really belongs to heads, a key feature of this system.

4.4 Theoretical Implications on Merge and Move

The goal of this section is to briefly discuss theoretical implications of our approach to Merge and Move.
In other words, we wish to ask the question: is it reasonable, from a theoretical perspective, to group all
specifier introduction together as Move, separating that from head-complement Merge? This section begins
to discuss this question, and argues that this sort of distinction is theoretically reasonable.

Consider the following English sentences:

(4) a. Sam seems to like flowers.

b. It seems Sam likes flowers.

Sentences (4a) and (4b) can be analyzed to be structurally related, with trees something like the follow-
ing\(^8\).

\(^8\)We will not discuss the finiteness difference between *to like flowers* in (4a) and *likes flowers* in (4b), except to suggest that
it may have something to do with there being no subject in the embedded clause in (4a) for the tense to assign nominative case.
\(^9\)We do not give these syntactic trees in full detail; for example, we do not give the internal structure of the VP or include
projections of C. This is for clarity of presentation.
The essential difference between these structures is that in (4a), the tense attracts the subject of the embedded clause, *Sam*, into its specifier, while in (4b), *Sam* is not attracted and instead a dummy subject *it* is inserted.

If we treat the introduction of *it* in (4b) as a Merge and the movement of *Sam* in (4a) as a Move, then we have an inelegant asymmetry between very similar structures. In contrast, if we consider both these specifier introductions to be Move operations, then the structures are appropriately parallel. The matrix tense attracts a DP, and that DP can be attracted from within the existing structure (as in the movement of *Sam* in (4a), a typical Move) or from elsewhere (as in the insertion of dummy *it* in (4b)).

Therefore, raising verbs provide an example of a case when Move (in the traditional sense) and addition of a new specifier are parallel. It is worth investigating further theoretical implications, as well as computational implications, of this way of dividing Merge and Move, as it provides a different way of understanding the
processes involved in minimalist derivations.

5 Conclusions

In this paper, we discussed current approaches to implementing minimalist syntax in a computational approach, focusing on the work by Fowlie and Koller (2017). Our main contribution is a modification of their minimalist grammar which takes the head-complement-specifier relationship, as well as the systematic differentiation between heads and phrases, as a built-in property of the system. In other systems, such properties can be derived from relationships such as dominance in the syntax tree, but in this system, the derivation goes the other way. This approach is well syntactically motivated, as heads and phrases exhibit syntactic differences, and may provide a convenient way to encode minimalist analyses in a computational system.

To illustrate this approach, we provide code which converts between derivation trees and minimalist trees, and can print both. At this point, this code cannot easily interface with any existing systems, including minimalist parsers; however, this is primarily a consequence of the fact that the specific structure to store derivation trees is unique to our system. It is likely that other structures designed to hold derivation trees could easily be converted to and from our derivation trees, as they hold essentially the same information; if this could be done, then our tool could be used to translate the output of a minimalist parser which produces derivation trees into actual objects (i.e. minimalist trees) in a minimalist grammar. Furthermore, the code we have implemented is close to what Fowlie and Koller (2017) call an interpretation into the minimalist grammar, which is vital to their parsing system; it could therefore be a step toward parsing this sort of minimalist grammar efficiently.

One major area not discussed is the relationship between this syntactic formalism and Lecomte (2008)’s minimalist categorial grammars, which use formal proofs to encode minimalist structures. It would be valuable to see how Lecomte’s categorial grammars would be modified to systematically encode the distinction between heads and phrases, much like the minimalist grammar given here.

Introducing a head-phrase distinction does not increase expressive capacity; in fact, it restricts possibilities, as it limits Merge to only head-complement relationships and Move to only head-specifier relationships. However, the distinction is highly linguistically motivated, and provides a useful way to computationally encode a minimalist analysis. For this reason, this approach could help computational approaches to minimalism better align with theoretical analyses, and therefore be convenient as a framework to build a computational minimalist grammar for a language.
References


