Implementing Minimalist Syntax and Remove

Johannes Englisch

Abstract

This article presents a computational implementation of Minimalist Syntax. The program implements a fragment of German grammar, building syntactic structure from a numeration and applying syntactic, such as Move, Agree, and Structure Removal. The article will also apply this system to two specific analyses: complex pre-fields and embedded clauses, which showcase the capabilities, but also the limitations of the program.

1. Introduction

This article describes a computer program that implements a fragment of German grammar within Minimalist Syntax (Chomsky 1995). This program simulates the application of Merge, Movement, and Agreement to generate syntactic structure. In addition to these rules, the implemented system also includes the concept of structure removal (Müller 2017).

The program imitates the generation of syntactic structures from numeration to final tree. This stands in contrast to, for instance, parsing algorithms, which extract syntactic structure from a string of surface symbols.[1] The program starts by taking out a vocabulary item from the numeration and using it as a starting point, from which it builds structure from the bottom up. And whenever there are multiple ways a derivation could progress, the program branches out and pursues all possible options.

Note that Minimalism itself does not require a specific starting point for a derivation: The rule system just generates all possible combinations of all items in the numeration. However, quite a few of these derivations result in the same tree structure. In fact, for every leaf node in a tree there is a derivation that 'started with this node'.[2] Spurious derivations like these do not pose a problem from a theoretical stand point, as they all apply the same operations to the

Structure Removal, 323–342
Andrew Murphy (ed.)
Linguistische Arbeits Berichte 94, Universität Leipzig 2019

[1] See Harkema (2001) for a parser based on Stabler (1997)'s formalisation of Minimalist syntax.

[2] Also, if you take structure removal into account, then there are derivations starting with an element that does not even show up in the final tree.

same elements and produce the same dependencies – just in a slightly different order. However, they do matter for computational implementations, since recalculating the same structure multiple times is a waste of both memory and computing time.

The article is structured as follows. Section 2 outlines what will and will not be part of the program and Section 3 illustrates how the program is structured from a bird's-eye view. Then, Section 4 describes the progression of a syntactic derivation from numeration to final tree. After this the article goes more into detail on how the program is implemented, with Section 5 showing how syntactic structure is represented with the program , Section 6 outlining the definitions of the actual syntactic rules, and Section 7 addressing some theoretical decisions made in the system. With the theoretical bits out of the way, Section 8 shows how the program behaves when applied to actual data. Finally, Section 9 concludes the article.

2. Scope of the program

As stated above, the program aims to model a fragment of German grammar. This fragment generates sentence structures and recreates common phenomena like wh movement, case assignment, and V2 word order. This generation process restricts itself to the purely syntactical part of generative grammar, meaning that the system does not attempt to implement variable binding, covert movement, clitic formation, or other phenomena living at the interface to phonology, morphology, or semantics.

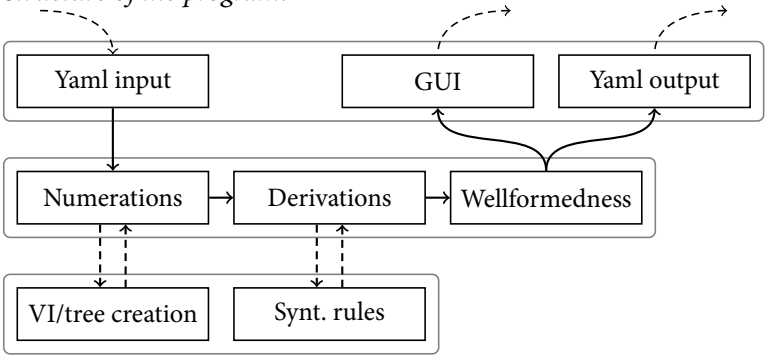
The program provides an implementation to the standard rules of Minimalist Syntax: Merge, Movement, Head Movement, and Agreement. In addition the program includes Müller (2017)'s structure removal process, which 'unbuilds' the syntactic structures created by Merge and Move.

Note that Minimalism does not define rules specifically tailored to any particular language. Instead the rules are kept very general and the difference between languages lies mainly in the feature specification of the vocabulary items in the lexicon. For instance, the V2 word order in German results from the C head bearing a feature that attracts the finite verb via head movement and another feature that triggers topicalisation, rather than from a language-specific movement rule or word order restriction.

3. Structure of the Program

As the diagram in (1) illustrates, the program is structured into three layers. The lowest layer contains the primitive rules of Minimalist Syntax and is responsible for the creation and manipulation of syntactic tree structure. The second layer lives in the realm of derivations, where numerations are created, derivations are performed, and tree structures are judged for their technical wellformedness. On top of all of that, there is the user interface. This layer parses the input provided by the user in the Yaml format[3] and displays the resulting derivations in a graphical user interface.

(1) *Structure of the program:*



Note that this structure isolates the rules of syntax from the rest of the system. A syntactic rule is merely a function mapping trees to trees. This means that their application is local to the structures directly involved in the process. Rules cannot refer to other elements in the work space or the numeration, nor can they access previous or future steps of the derivations, let alone different paths the derivations could have taken.

4. From numeration to derivation

Every computer program known to man turns some input into some output. The input of this program is a numeration of lexical items and a feature specification for the item the derivation should start with. The output consists a list of all derivations with a report of the technical wellformedness of the resulting tree attached to them.

[3] `http://yaml.org/`

The program starts by looking through the numeration for any items that qualify as a starting point. These items are put into an agenda, together their own copy of the numeration. Then the program takes the tree–numeration pairs out of the agenda one by one and tries to apply the rules of syntax to them. The results of successful rule applications are pushed back onto the agenda to keep the derivation going, including a reference to the previous state of the derivation, so the process can be retraced later. If none of the rules are successful the tree–numeration pair is added to a list of final structures. This process repeats until the agenda is empty.

After finishing all derivations the program checks the technical well-formedness their final structures. It considers the final state of a derivation grammatical if:

- There are no elements left in the numeration,
- There are no unsatisfied operation triggers left in the structure, *and*
- There are no unvalued features left in the structure.

5. Representation of syntactic structure

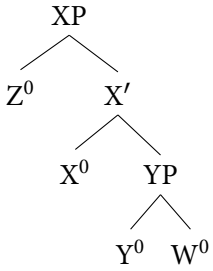
Before we can see how syntactic structure is manipulated, we need to take a look at how these structures are represented in the first place. This section looks at the representation of phrases (Section 5.1), heads (Section 5.2), and whole derivations (Section 5.3).

5.1. Syntactic phrases

Syntactic structure is represented in terms of *phrases*. Every phrase consists of a head and a list of arguments as shown in (2a). This representation can be converted back and forth to a binary syntax tree, as illustrated in (2b). Note that X′ and XP nodes of a tree are not represented directly in the data. These nodes are created during the conversion based on the head. This gives the system a simple account for projection: Because all dominant nodes are made from the same head, any changes to that head happen to the whole projection line. However, this makes it impossible for syntactic rules to target only parts of a projection. This directly affects the way head movement works in the program (see Section 6.3).

(2) a. *Representation of a phrase:*
{ head: X^0,
args: [{ head: Z^0, args: [ ] },
{ head: Y^0, args: [{head: W^0, args: [ ]}] }] }

b. *Corresponding tree:*



5.2. Syntactic heads

Heads like in (3a) are represented as shown in (3b). A head consists of three parts: A phonological representation, a set of morpho-syntactic features and an ordered list of operation-triggering features. Morpho-syntactic features are a set of key–value pairs, although the value may be empty, which is indicated with the placeholder symbol *e*. Operation-triggering features are uninterpretable features, which require the application of syntactic rules. They are tuples of the type of operation, a feature the target must carry, and – if necessary – the feature value the target must match.

(3) a. 'do' [cat:T, case:nom, ϕ:□, •cat:v•≫∗case∗≫∗ϕ∗≫•cat:D•]

b. *Representation of 'do':*
{ phon: 'do',
features: { 'cat': 'T', 'case': 'nom', 'phi': e },
triggers: [(p-select, 'cat', 'v'),
(agree, 'case'),
(agree, 'phi'),
(p-select, 'cat', 'D')] }

5.3. Derivations

In this program a derivation is merely a list of discrete steps, as shown in (4). Each step consists of the current numeration and the syntactic structure built so

far. Again, this structure is the point that allows the user to look at intermediate steps of a derivation and observe the application of rules, whose effects are not directly visible in the final structure – i. e. structure removal.

(4) *Representation of a derivation:*
[{ numeration: {*book*, *read*},
tree: *a* },
{ numeration: {*read*},
tree: [*a book*] },
{ numeration: { },
tree: [*read* [*a book*]] }]

6. Implementation of syntactic rules

This section looks at the implementation of Merge (Section 6.1), movement (Section 6.2), head movement (Section 6.3), agreement (Section 6.4), and structure removal (Section 6.5). As mentioned above, these rules are merely functions that take a tree – or two, in the case of Merge – as arguments and return a transformed tree structure.

6.1. Merge

Merge builds structure by combining two smaller trees into a bigger one. It is caused by a selection feature [•F:v•], which is deleted in the process. In the actual implementation, Merge adds the target to the argument list of the phrase that triggers the operation. This process is illustrated in (5).

(5) a. *Before Merge:*
{ head: { phon: 'read',
features: {'cat': 'V'},
triggers: [(p-select, 'cat', 'D')] },
args: [ ] }

b. *After Merge:*
{ head: { phon: 'read',
features: {'cat': 'V'},
triggers: [ ] },
args: [{ **head: { phon: 'it',**
features: {'cat': 'D'},
triggers: [ ] },
**args: [ ] }] } }

6.2. Phrasal movement

Phrasal movement merges a tree with its own subtree and replaces the starting position of the subtree with emptiness *e*. This process is illustrated in (6). To reflect their conceptual similarities, Move and Merge are both triggered by selection features. This gives rise to situations where a derivation could both merge and move. In these cases the system favours Merge.

(6) a. *Before Move:*
{ head: { phon: 'x',
features: {'cat': 'X'},
triggers: [(p-select, 'cat', 'Y')] },
args: [{ head: { phon: 'y', features: {'cat': 'Y'}, triggers: [ ] },
args: [ ] },
{ head: { phon: 'z', features: {'cat': 'Z'}, triggers: [ ] },
args: [ ] }] }

b. *After Move:*
{ head: { phon: 'x',
features: {'cat': 'X'},
triggers: [(p-select, 'cat', 'Y')] },
args: [**e**,
{ head: { phon: 'z', features: {'cat': 'Z'}, triggers: [ ] },
args: [ ] },
{ head: { phon: 'y', features: {'cat': 'Y'}, triggers: [ ] },
**args: [ ] }] }

This raises the question how the algorithm finds a target for movement. It starts with the complement – i. e. the first member of the argument list – of the top-most phrase and checks if it meets the requirements of the selection feature. If that fails the algorithm descends down into the complement, checks its arguments from highest to lowest, and keeps doing so recursively until

it either finds a suitable target or reaches the bottom of the tree. Note that, at the present state of the system, movement is not restricted by any locality constraints.

6.3. Head movement

Head movement, on the other hand, merges the *head* of a tree with the head of its own subtree. It is triggered by a head selection feature [•F^0•] and moves the head of a lower phrase onto the top-most head. However, remember that any change of the head affects the whole projection line. This means that removing the head with all its features, would render the remaining phrase inaccessible for further agreement or movement operations. The program avoids this effect by moving only the *phonological representation* of the head, leaving the features in place, as shown in (7). Also note that this implementation of head movement obeys the Head Movement Constraint (Travis 1984), according to which a head may move only to the next dominating phrase, but not higher.

(7) a. *Before head movement:*

```
{ head: { phon:     'x',
          features: {'cat': 'X'},
          triggers: [ (h-select 'cat' 'X') ] },
  args:  [ { head: { phon:     'y',
                    features: {'cat': 'Y'},
                    triggers: [ ] },
             args:  [ ] } ] }
```

b. *After head movement:*

```
{ head: { phon:     ('x', 'y'),
          features: {'cat': 'X'},
          triggers: [ ] },
  args:  [ { head: { phon:     e,
                    features: {'cat': 'Y'},
                    triggers: [ ] },
             args:  [ ] } ] }
```

6.4. Agreement

Agreement ensures that two heads share the same value for a feature. There are two main implementations of agreement. In *Feature checking* the numeration

provides fully specified heads. Agreement compares two heads with respect to a given feature and crashes the derivation if their values differ. In *Feature valuation* the heads in the numeration may contain features without a value. Agreement fills these gaps by copying values from other heads. When the derivation is over, any unvalued feature left renders the syntactic structure ungrammatical. This system uses feature valuation, mainly because this allows the program to trim number of derivations.

Take, for instance, a sentence with two DPs bearing different number features. In feature checking the numeration contains two determiners: a plural one and a singular one. As (8) shows, the grammar generates two structures, one of which crashes because the determiners were merged with 'incompatible' nouns.

(8) *Feature checking:*

a. dass das Kind die Bücher liest
that the.[num:sg] child.[num:sg] the.[num:pl] books.[num:pl] reads

→ Success

b. *dass die Kind das Bücher liest
that the.[num:pl] child.[num:sg] the.[num:sg] books.[num:pl] reads

→ Crash

In feature valuation the numeration contains two instances of *one* general determiner. This means that that structure building can only produce the one derivation shown in (9). Merge attaches the two instances to the nouns and agreement fills their number features from their context. The program does not need to compute a second, failing derivation.

(9) *Feature valuation:*

dass das/die Kind das/die Bücher liest
that the.[num:□] child.[num:sg] the.[num:□] books.[num:pl] reads

→ Success

Note, however, that feature valuation struggles with cases like case concord. While feature checking can establish agreement locally within a DP, feature valuation requires a way for the case assigner to agree with multiple heads. For this the system would either have to add more probes or a mechanism for mass-valuing DPs.

Agreement is triggered by a probe feature [∗F∗]. The head bearing a probe

can only look downwards, when searching for a target. The search algorithm mirrors that of Move, the difference being that Agree also looks at its own specifiers for targets. Once agreement finds a target it can copy the feature value in either direction – the example in (10) illustrates downwards valuation. Because of this a T head can both value its ϕ features from a lower target and copy its case features downwards to assign case. Note that this also means that a case assigner must itself carry case.

(10) a. *Before agreement:*

```
{ head: { phon:     'x',
          features: {'cat': 'X', 'F': 'val'},
          triggers: [ (agree, 'F') ] },
  args:  [ { head: { phon:     'y',
                    features: {'cat': 'Y', 'F': e},
                    triggers: [ ] },
             args:  [ ] } ] }
```

b. *After agreement:*

```
{ head: { phon:     'x',
          features: {'cat': 'X', 'F': 'val'},
          triggers: [ ] },
  args:  [ { head: { phon:     'y',
                    features: {'cat': 'Y', 'F': 'val'},
                    triggers: [ ] },
             args:  [ ] } ] }
```

6.5. Structure removal

As the name suggests, Remove deletes the argument of a head. Like movement the operation is split into a phrasal and a head version. In the case of phrasal Remove, a head bearing a Remove feature (−F−) looks among its arguments for the highest instance of the feature F and deletes the argument. Note that the *entire* argument is removed, as shown in (11).

(11) a. *Before phrase removal:*
{ head: { phon: 'x',
features: {'cat': 'X'},
triggers: [(p-remove, 'cat', 'Y')] },
args: [{ head: { phon: 'y',
features: {'cat': 'Y'},
triggers: [ ] },
args: [ ] }] }

b. *After phrase removal:*
{ head: { phon: 'x',
features: {'cat': 'X'},
triggers: [ ] },
args: **[ ]** }

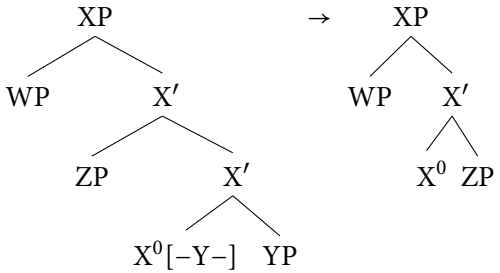
Head Removal, on the other hand, deletes only the head of an argument. It is triggered by a Head Remove feature ($-F_0-$). Not that, unlike head movement, head removal also affects the dominating X′ and XP nodes. This means that the whole projection line of the argument disappears, and any elements attached to that projection line need to be reintegrated into the structure. It does so by attaching these elements as arguments of the head that triggered head removal, in place of the removed element.

(12) a. *Before head removal:*
{ head: { phon: 'x',
features: {'cat': 'X'},
triggers: [(h-remove, 'cat', 'Y')] },
args: [{ head: { phon: 'y',
features: {'cat': 'Y'},
triggers: [ ] },
args: [{ head: { phon: 'a',
features: {'cat': 'A'},
triggers: [ ] },
args: [ ] }] }] }

b. *After head removal:*
{ head: { phon: 'x',
features: {'cat': 'X'},
triggers: [ ] },
args: [{ **head: { phon: 'a',**
features: {'cat': 'A'},
triggers: [ ] },
**args: [ ] }] }

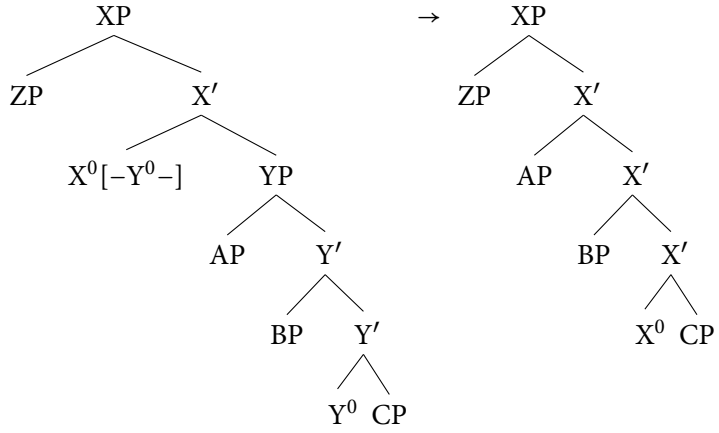
Due to its destructive nature, structure removal can change how future rules treat the arguments of a phrase. On one hand, this happens when removal applies to a complement. In the case of phrase removal the whole complement disappears and the lowest specifier takes its place, as one can see in (13). This also means that from this point on all syntactic rules will treat that element as a complement.

(13)



On the other hand, head removal can cause a similar effect when deleting a complement and reintegrating its arguments. As the example in (14) illustrates, only the lowest argument attaches to the complement position, while the other arguments land in specifier positions. Again, this means that these elements take on the syntactic properties of specifiers. This gives the system a second way of shifting elements up a tree besides movement.

(14)



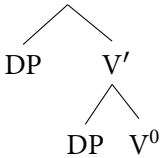
7. Brief theoretical considerations

This section briefly addresses two topics that have been left out of the discussion thus far: First, Section 7.1 looks at the role of the linear order of elements within the program. Then, Section 7.2 looks at how the system implements – or rather *does not* implement – c-command.

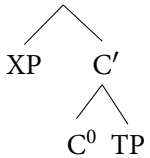
7.1. A word on linearisation

The program linearises the structure post-syntactically, meaning that above rules are entirely unaffected by the linear order of any elements. On a technical level this is done using rules which specify the ordering of a head with its complement or its specifiers. Which rule is applied to a phrase depends on the lexical category on the head. The example in (15) shows two examples of such rules in German.

(15) a. Verb phrase
Rule: { cmpl: left,
spec: left }
Tree:



b. Complementiser phrase
Rule: { cmpl: right,
spec: left }
Tree:



7.2. The role of c-command

At the moment there is no definition of c-command in the program. The reason for that is that it is not really needed for the given data set. In movement a landing site c-commands its starting position, because of the way movement attaches a moved element as a specifier at the top. Agreement, on the other hand, looks for the highest target that is dominated by the root node. However, this is likely to be a property of the specific data set covered by the program rather than a general principle. Some of the rules left out from the grammar fragment, like variable binding, may still be sensitive to c-command.

8. Running the program on actual analyses

This section contains two example analyses to illustrate the program more practically. Section 8.1 re-implements the remove-based analysis of double prefields by Müller (2017) to show a case, where the program progresses as expected. On the flip side, Section 8.2 goes through a derivation involving embedded clauses to show some limitations of the system. After that, Section 8.3 looks at some of the possible causes for said limitations.

8.1. A success story: double prefields

As mentioned above, the first example analyses double prefields using structure removal. This is a direct adaptation of an analysis by Müller (2017), which serves as empirical evidence for head removal. The object of the study are sentences like in (16). In this case we have a German V2 structure, except that the verb seems to be positioned in the *third* position. To be more accurate, Müller (2017:21–24) notes that the elements in the prefield show both signs of being multiple constituents as well as one single unit.

(16) [$_{DP}$ Den Fahrer] [$_{PP}$ zur Dopingkontrolle] begleitete ein
the rider.ACC to.the doping.test accompanied a
Chaperon
chaperon.NOM
'A chaperon accompanied the rider to the doping test'

The world of structure removal answers to conflicting evidence by taking advantage of the derivational nature of the system: You can have your cake first

and eat it later. In other words, the elements in the prefield start off as part of *one* constituent, which gets destroyed at a later step of the derivation; after the V2 structure has been build.

The analysis achieves this by first creating a V2 structure and then dismantling parts of it using head removal. Such a derivation is illustrated in (17): First the verb is head-moved out of the VP into the C head. Then the whole remaining verb phrase is moved to Spec-CP via topicalisation, creating a V2 structure. Finally, head removal destroys the VP projection of the topicalised verb phrase and the two arguments are reintegrated as specifiers of the CP.

(17) a. chaperon [$_{VP}$ rider to.the.test accompany] | h-move
b. accompany [chaperon [$_{VP}$ rider to.the.test t_{V^0}]] | topicalise
c. [$_{VP}$ rider to.the.test t_{V^0}] [accompany [chaperon t_{VP}]] | remove
d. rider to.the.test [accompany [chaperon t_{VP}]]

The concrete implementation performs the derviation based the numeration listed in (18). Note that the two instances of the determiner *the* are collapsed into one entry in the numeration.[4]

(18) *Numeration:*
a. 1× ∅ [cat:C, •cat:T•≫•cat:T^0•≫•top:t•≫−top:t^0−]
b. 1× ∅ [cat:T, case:nom, •cat:v•≫•cat:v^0•≫∗case∗]
c. 1× ∅ [cat:v, case:acc, •cat:V•≫∗case∗≫•cat:D•≫•cat:V^0•]
d. 1× begleiten [cat:V, top:t, •cat:P•≫•cat:D•]
e. 1× zu [cat:P, case:dat, •cat:D•≫∗case∗]
f. 2× das [cat:D, case:□, •cat:N•]
g. 1× Dopingkontrolle [cat:N]
h. 1× Fahrer [cat:N]
i. 1× ein [cat:D, case:□, •cat:N•]
j. 1× Chaperon [cat:N]

Since the program is unable to take semantics into account, it will blindly generate all possible combinations of arguments and determiners. This results in 18 sentences such as the ones in (19), all of which show the desired double-prefield construction.

[4] Also, note that for the sake of clarity this analysis omits some non-essential syntactic processes such as ϕ agreement or the contraction of the preposition *zu* with the determiner.

(19) a. [$_{DP}$ den F.] [$_{PP}$ zu der DK.] begleitete ein Ch
b. [$_{DP}$ den F.] [$_{PP}$ zu einer DK.] begleitete der Ch
c. [$_{DP}$ einen F.] [$_{PP}$ zu der DK.] begleitete der Ch
⋮
q. [$_{DP}$ den Ch.] [$_{PP}$ zu einer DK.] begleitete der F
r. [$_{DP}$ einen Ch.] [$_{PP}$ zu der DK.] begleitete der F

8.2. A problematic story: V2 and embedded clauses

The second example derives the seemingly simple example in (20). The sentence consists of an embedded verb-final sentence with a complementiser and a matrix clause with a verb-second word order. It poses a challenge, since the material in the numeration needs to be correctly merged and moved across two clauses. The numeration is shown in (21).

(20) Tobi weiß [dass Silke schläft]
Tobi knows that Silke sleeps

(21) *Numeration:*
a. 1× ∅ [cat:C, •cat:T•≫•cat:T^0•≫•top:t•]
b. 2× ∅ [cat:T, case:nom, •cat:v•≫•cat:v^0•≫∗case∗]
c. 2× ∅ [cat:v, •cat:V•≫•cat:D•≫•cat:V^0•]
d. 1× Tobi [cat:D, top:t, case:□]
e. 1× weiß [cat:V, •cat:C•]
f. 1× dass [cat:C, •cat:T•]
g. 1× Silke [cat:D, case:□]
h. 1× schläft [cat:V]

For this numeration the program generates four distinct derivations as in (22), which are all judged to be well-formed on a technical level. However, they all showcase some challenges for the system.

(22) a. *dass Silke [Tobi schläft] weiß
b. *Tobi weiß Silke, [dass schläft]
c. *dass Tobi [schläft Silke] weiß
d. Tobi weiß [dass Silke schläft]

The first ungrammatical example is the result of the C heads being merged in the wrong order. The phonologically empty C head of the matrix clause

happened to get merged *first*, ending up in the embedded clause. This leads to a V2 clause embedded within the middle field of a verb-final clause.

(23) *dass Silke [Tobi$_1$ schläft t_1] weiß
that Silke Tobi sleeps knows

A possible repair mechanism for this sentence is extraposition. However, whether or not extraposition yields a grammatical structure depends on the matrix verb, as shown in (24).

(24) a. *dass Silke weiß, [Tobi schläft]
that Silke knows Tobi sleeps
b. dass Silke glaubt, [Tobi würde schlafen]
that Silke believes Tobi would sleeps
'that Silke thinks that Tobi would sleep'

The second successful derivation shown in (25) is also a false positive. In this case the DP with the [topic] feature is merged within the embedded clause and then topicalised to the specifier of the matrix CP. This arises due to the lack of locality constraints within the implemented system.

(25) *Tobi$_1$ weiß Silke [dass t_1 schläft]
Tobi knows Silke that sleeps

The example in (26) is the result of the double duty selection features need to fulfil. This derivation merges the V2-triggering C head and the DP *without* the [topic] feature in the embedded clause. This means that the C head topicalisation lacks a target to topicalise. However, since selection features trigger both Move *and* Merge, the derivation turns to the numeration and merges the other DP. After this the topicalised argument gets moved into the matrix vP.

(26) *dass Tobi$_1$ [t_1 schläft Silke] weiß
that Tobi sleeps Silke knows

The last derivation found by the system is shown in (27). On the surface this example looks perfect. The verb-final clause is embedded within the V2 structure and all the arguments are merged and moved in the right places.

(27) Tobi weiß t_1 [dass Silke schläft]
Tobi knows that Silke sleeps

However, this result seems quite fragile. Consider a variant of the same numeration, where the T heads bear a [•D•] feature causing EPP movement. Such a configuration results in the derivation shown in (28). Just like in (26), the selection feature on the embedded T head can trigger both Merge and Move. And while T has a valid target for movement, it also has a valid target for Merge, resulting in the matrix subject to be merged too early. In a later step the subject moves into the matrix clause, resulting in a structure which is string-invariant to (27). In other words the stucture looks correct on the surface, but is *structurally* wrong.

(28) *[$_{CP}$ Tobi$_1$ weiß [$_{TP}$ t_1 [$_{vP}$ t_1 [$_{CP}$ dass [$_{TP}$ t_1 [$_{vP}$ Silke schläft]]]]]]
Tobi$_1$ knows that Silke sleeps

8.3. Analysing the undesired effects

Now, let us take a look at the potential causes for the behaviour of the program in section 8.2. As mentioned above, the omission of any locality constraints allows for unrestrained long-distant movement. Note, however, that this is a property of the *implementation* rather than the grammatical formalism.

Another source for error lies in the tight connection of Merge and Move and their triggers. Potentially a [•F•] feature can always cause either Merge *or* Move. The system resolves this by choosing Merge over Move, when a conflict arises. However, this ties the application of Move to the current state of the numeration:

- Movement can be blocked by elements in the numeration intended to be merged in other clauses.
- A derivation failing to merge a movement target in time for the operation can always be repaired by base-generating directly from the numeration.

This problem is amplified by the fact that Minimalism does not implement the concept of 'clause boundaries' on a technical level. Clauses are usually CPs, but the rule system in itself treats C like any other category.[5] This turns the

[5] This includes the Phase Impenetrability Condition, which defines phases in terms of *phrases*, not clauses (CP and VP in the case of Chomsky (2000:106–107), or any and every phrase in Müller (2003)).

numeration into one big bag containing all elements of all clauses, each of which could potentially bleed or feed movement operations throughout the whole derivation.

9. Conclusion

In conclusion, this article presented a program which imitates the progression of syntactic derivations within the Minimalist framework. The program succeeds in building phrase structure, including the application of movement and structure removal. However, the system reaches its limits, when the resulting structure spans across one or more clause boundaries, which might be rooted in the fact that neither the rules of grammar, nor the representation of its structures are structured in terms of clauses or clause boundaries.

References

Chomsky, Noam (1995). *The Minimalist Program*. Vol. 28. of *Current Studies in Linguistics*. MIT Press: Cambridge, MA.

Chomsky, Noam (2000). Minimalist Inquiries: The Framework. In R. Martin, D. Michaels & J. Uriagereka (eds). *Step by Step: Essays on Minimalist Syntax in Honor of Howard Lasnik*. MIT Press: Cambridge, MS. 89–155.

Harkema, Hendrik (2001). *Parsing Minimalist Languages.* PhD thesis, UCLA. Los Angeles

Müller, Gereon (2003). Phrase Impenetrability and Wh-Intervention. In A. Stepanov, G. Fanselow & R. Vogel (eds). *Minimality Effects in Syntax*. de Gruyter: Berlin. 289–325.

Müller, Gereon (2017). Structure Removal: An Argument for Feature-Driven Merge. *Glossa: A Journal of General Linguistics* 2(1). 1–35.

Stabler, Edward (1997). Derivational Minimalism. In C. Retoré (ed.). *Logical Aspects of Computational Linguistics: First International Conference, LACL '96, Nancy, France, September 23–25, 1996*. Vol. 1328. of *Lecture Notes in Artificial Intelligence*. Springer-Verlag: Berlin. 68–95.

Travis, Lisa (1984). *Parameters and Effects of Word Order Variation.* PhD thesis, MIT. Cambridge, MA