Computational implementation of a top-down parser-grammar

Program version 1.9x

Pauli Brattico

IUSS, Pavia

Abstract

This document describes a computational implementation of a minimalist top-down parser-grammar. The parser-grammar is based on the assumption that the core computational operations of narrow syntax (e.g., Merge/Move/Agree) are applied incrementally in language comprehension. Full formalization with a description of the Python implementation will be discussed, together with a few words towards empirical justification. The theory is assumed to be part of the human grammatical competence (UG).

Table of Contents

1 Introduction

Language sciences distinguish linguistic competence from linguistic performance (Chomsky 1965). Competence refers to the knowledge of language possessed by native speakers. Any native speaker can answer questions such as “Is that sentence grammatical?” or “What does this sentence mean?” and, by doing so, provide a rich body of knowledge about his or her language. One might want to take a further step and examine also linguistic performance, how speakers and hearers put that knowledge in use in real-time communication. What kind of mistakes do they make? How do they recover from errors? The latter inquiry, however, presupposes the former: a theory of language use requires an explicit or implicit theory of the language whose use is being investigated. Competence, on the other hand, can be studied without taking performance into account.

It does not follow from this that facts pertinent to linguistic performance must be irrelevant to the theory of competence. Whether performance should be taken into account in a theory of competence can only be answered by an empirical inquiry. One strong (and hence interesting) form of a theory which assumes that performance should be taken into account when formulating a theory of competence is that of (Phillips 1996), who assumes that the theory of competence must incorporate the *parser*, a computational system that generates (correct) syntactic representations for linguistic input available to the hearer. Whether language comprehension/parsing is or is not part of the theory of linguistic competence is an empirical question that cannot be settled by conceptual or methodological reasoning alone. I will argue, following Phillips, that certain aspects of performance do constitute an important part of the theory of competence. Specifically, the main hypothesis pursued in this study is that the “one-dimensional” properties of the PF-interface, when looked from the perspective of language comprehension, matter for the theory competence.

This document consists of two parts. The first, Section 2, details the empirical framework assumed as a background in this study. These are assumptions that are, in my view, justified both by empirical and practical concerns; they constitute the hypothesis put forward in this document. The second part, Section 3, goes into details concerning full formalization and the computational implementation of that formalization. The claims made in these sections have a different status than those made in the first section. First, many decisions that were made concerning formalization could be done in several different ways, thus there is a certain degree of arbitrariness when it comes to many of the formal assumptions presented here. Second, many of these assumptions were are still are justified on the basis of concrete simulations: something went

wrong, and a correction was required. That correction shows here in a form of detailed assumptions of some computational operation that might strike the reader as unmotivated and/or unjustified. When that is the case, then it is important to realize that any assumption presented was motivated by some concerns or error in the concrete simulation, and that the package of assumptions presented here constitutes, when taken together, a system that at least does work correctly in connection with a corpus of sentences and/or some empirical phenomenon that was being explored at the time. Finally, the ultimate formalization, the computer code itself, is in many ways uneconomic and contains pieces of code that require refactoring; the reason that was not done is that any change in the program could potentially cause a change in its behavior in relation to bulk of empirical data, and thus there comes a point in any research project at which the program cannot be changed anymore for replication reasons. Dramatic changes are only possible after a certain project has been completed and the code that was used in it stored and forked away.

2 The framework

2.1 Merge

Linguistic input is first received by the hearer in the form of sensory stimulus. That input can be thought abstractly as a one-dimensional, linear string α, β, … of phonological words. We assume that in order to understand what the sentence means the human parser must create a set of abstract syntactic interpretations for the input string received through the sensory systems. These interpretations and the corresponding representations might be lexical, morphological, syntactic and semantic. One fundamental concern is to recover the hierarchical relations between words. To satisfy this condition, let us assume that while the words are consumed from the input, the core recursive operation in language, Merge (Chomsky 1995, 2005, 2008), arranges them into a hierarchical representation. For example, if the input consists of two words α + β, Merge will yield [α, β](1).

(1) John + sleeps.
↓ ↓
[John, sleeps]

The assumption that this process is incremental means that each word consumed from the input will be merged to the phrase structure as it is being consumed. For example, if the next word is *furiously*, it will be merged with (1). There are three possible attachment sites, shown in (2), all which correspond to different hierarchical relations between the words.

(2) a. [[John *furiously*] sleeps] b. [[John, sleeps] *furiously*] c. [John [sleeps *furiously*]]

Several factors regulate this process. One concern is that the operation creates a representation that is in principle ungrammatical and/or uninterpretable. Alternative (a) can be ruled out on such grounds: *John*

furiously is not an interpretable fragment. Another problem of this alternative is that it is not clear how the adverbial, if it were originally merged inside subject, could have ended up as the last word in the linear input. The default linearization algorithm would produce **John furiously sleeps* from (2)a. Therefore, this alternative can be ruled by the fact that the result is ungrammatical and is not consistent with the word order discovered from the input. But what *is* consistent with the linear ordering?

If the default linearization algorithm proceeds recursively in a top-down left-right order, then each word must be merged to the "right edge" of the phrase structure, "right edge" referring to the top node and any of its right daughter node, granddaughter node, recursively. See Figure 2.

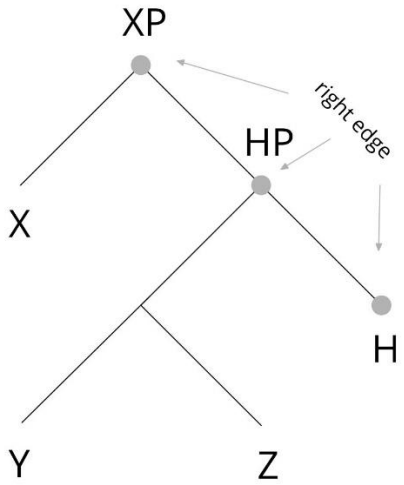


Figure 2. The right edge of a phrase structure.

This would mean either (b) or (c). (Phillips 1996) calls the operation "Merge Right". We are therefore left with the two options (b) and (c). The parser-grammar will select one of them. Which one? There are many situations in which the correct choice is not known or can be known but is unknown at the point when the word is consumed. In an incremental parsing process, decisions must be made concerning an incoming word without knowing what the remaining words are going to be. We can experience this situation when reading a garden-path sentence: a wrong decision leads into an error that is recognized when reaching the end (as in *The horse raced past the barn fell*). It must be possible to backtrack and re-evaluate a parsing decision made at an earlier stage of the process. Let us assume that all legitimate merge sites are ordered and that these orderings generate a recursive search space that the parser-grammar can used to backtrack. The situation after consuming the word *furiously* will thus look as follows, assuming an arbitrary ranking:

(3) *Ranking*

a. [[~~John *furiously*], sleeps]~~ (Eliminated)

b. 1. [[John, sleeps] *furiously*] (Priority high)

c. 2. [John [sleeps *furiously*]] (Priority low)

The parser-grammar will merge *furiously* into a right-adverbial position (b) and branch off recursively. If this solution will not produce a legitimate output, it will return to the same point and try solution (c). Every decision made during the parsing process is treated in the same way.

Both solutions (a) and (c) are "countercyclic": they extent the phrase structure at its right edge, not at the highest node. A countercyclic operation is more complex than simple merge operation that combines to constituents: it must insert the constituent into a phrase structure and update the constituency relations accordingly (Section 3.4.1). This type of derivation is often called "top-down," because it seems to extend the phrase structure from top to bottom. The characterization is misleading: the phrase structure can be extended also in a bottom-up way, for example, by merging new items always to the highest node. It is more correct to say that merge is "to the right edge," whether up or down.

A final point that merits attention is the fact that merge right can break constituency relations established in an earlier stage. This can be seen by looking at representations (1) and (2)c, repeated here as (4).

(4) John sleeps furiously

↓ ↓ ↓

[John, sleeps] → [John [sleeps furiously]]

During the first stage, the words *John* and *sleeps* are sisters. If the adverb is merged as the sister of *sleeps*, this is no longer true: *John* is now paired with a complex constituent [*sleeps furiously*]. If a new word is merged with [*sleeps furiously*], the structural relationship between *John* and this constituent is diluted further (5).

(5) [John [[sleeps furiously] γ]

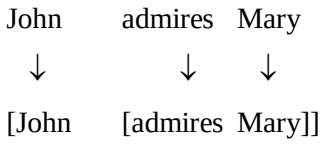
This property of the Phillips architecture has several important consequences. One consequence is that upon merging two words as sisters, we cannot know if they will maintain a close structural relationship in the derivation's future. In (5), they don't: future merge operations broke up constituency relations established earlier and the two constituents were divorced. Consider the stage at which *John* is merged with the verb 'sleep' but with wrong form *sleep*. The result is a locally ungrammatical string **John sleep*. But because constituency relations can change in the derivation's future, we cannot rule out this step as ungrammatical. It is possible that the verb 'sleep' ends up in a structural position in which it bears no meaningful linguistic relation with *John*, let alone one which would require them to agree with each other. Only those configurations or phrase structure fragments can be checked for ungrammaticality that *cannot* be tampered in the derivation's future. Such fragments are called *phases* in the current linguistic theory (Chomsky 2000,

2001). I will return to this topic in Section 2.3. For present purposes, it is important to keep in mind that in the Phillips architecture constituency relations established at point t_1 do not necessarily hold at a later point t_2.

As this discussion shows, the language comprehension perspective requires that we make several nontrivial and nonstandard assumptions concerning Merge. I do not know any way to avoid these (or some very similar) assumptions if we assume that language comprehension is incremental and that it uses the core computational operations of the human language. There are at least two ways to think about what these changes mean. The strong hypothesis, assumed by Phillips, is to say that parsing = grammar, hence that these are the true properties of Merge and nothing else is. We would then have to give up the standard properties of Merge that are postulated on the basis of the bottom-up production theories (e.g., strict cyclicity). A weaker hypothesis is that the theory of Merge must be consistent with these properties. According to this alternative, Merge must be able to perform the computational operations described above (or something very similar), but we resist drawing conclusions concerning the production capacities that constitute the basis of standard theories of competence. Perhaps Merge can operate in a production "mode" and in comprehension "mode". The weaker hypothesis is less interesting than the strong one, and possibly wrong in being less parsimonious, but it makes it possible to pursue the performance perspective without rejecting linguistic explanations that have been crafted on the basis of the more standard production framework; instead, we try to see if the two perspectives eventually "converge" into some core set of assumptions. This is the position taken in this work.

2.2 The lexicon and lexical features

Consider next a transitive clause such as *John admires Mary* and how it might be derived under the present framework (6).

(6)


There is much linguistic evidence that this derivation generates the correct hierarchical relations between the three words. The verb and the direct object form a constituent that is merged with the subject. We can imagine that this hierarchical configuration is interpretable at the LF-interface, with the usual thematic/event-based semantics: Mary will be the patient of the event of admiring, whereas John will be the agent. If we change the positions of the arguments, the interpretation changes. But the fact that the verb *admire*, unlike *sleep*, can take a DP argument as its complement must be determined somewhere. Let us assume that such facts are part of the lexical items: *admire*, but not *sleep*, has a *lexical selection feature* [!COMP:D] (or alternatively subcategorization feature) which says that it is compatible with, and in fact requires, a DP-complement. The idea has been explored by (Chesi 2004) within the framework of a top-down grammar. The

fact that *admire* has the lexical feature [!COMP:D] can be then used by the merge right operation to create a ranking: when *Mary* is consumed, the operation checks if any given merge site allows the operation. In the example (7), the test is passed: the label of the selecting item matches with the label of the new arrival.

(7)

John	admires	Mary	
↓	↓	↓	
[John	[admires	Mary]]	
	[COMP:D]	D	✓ Merge site accepted.

Feature [COMP:L] means that the lexical item *allows* for a complement with label L, and [!COMP:L] says that it *requires* a complement of the type L. Correspondingly, [–COMP:L] says that the lexical item does *not allow* for a complement with label L.

There is a certain ambiguity in how these features are used. When the parser-grammar is parsing an input, it will use these features to rank the solutions. They cannot be normally used to filter out anything, because the constituency relations could change in the derivation's future. But when the phrase structure has been completed, and these is no longer any input to be consumed, the same features can be used for filtering purposes. A functional head that requires a certain type of complement (say v-V) will crash the derivation if the required complement is missing while no more words are being consumed. This procedure concerns features that a positive and mandatory (e.g. [!COMP:V]) or negative ([–COMP:V]). This filtering operation is performed at the LF-interface and will be later called an "LF-legibility test." It purpose, if you will, is to make sure that the phrase structure can be interpreted by the semantic or conceptual-intentional systems.

Let us return to the example with *furiously*. What might be the lexical features that are associated with this item? The issue depends, of course, on the specific assumptions of the theory of competence, but let us assume something for the sake of the example. There are three options in (7): (i) complement of *Mary*, (ii) right constituent of *admires Mary*, and (iii) the right constituent of the whole clause. We can rule out the first option by assuming (again, for the sake of example) that a proper name cannot take an adverbial complement (*Mary* has a lexical selection feature [–COMP:ADV]. We are left with the two options. Let us examine the two options (8).

(8)

a.	[[$_S$ John	[admires	Mary]]	furiously]
b.	[John	[$_{VP}$ admires	Mary]	furiously]]

Completely independently of which one of these two solutions is the more plausible one (or if they both are equally plausible), we can guide Merge by providing the adverbial with a lexical selection feature which determines what type of *specifiers* (left phrases) it is allowed or required to have. I will call such features *specifier selection features*. A feature [SPEC:S] favors solution (a), whereas [SPEC:V] favors solution (b).

Analogously with the complement selection features, we use "!" to indicate that the selection is mandatory. If the adverbial has both feature or neither, then the selection is arbitrary. More generally then, suppose the item to be is merged is α; the specifier features of α will tell what type of complex phrases LP α can be merged with. The term "specifier" is here used slightly differently from the standard usage, but it is not misleading because in most cases the selected phrase [LP, α] will end up constituting a specifier of the element α that is merged. An accurate but also cumbersome name would be "left sister selection feature." This name comes with the caveat that since phrase structure configurations can change at a later stage of the derivation, what is a left sister at stage t can fail to be the left sister at some later stage.

2.3 Phases and left branches

Let us consider next the derivation of a slightly more complex clause (9).

(9) John's mother admires Mary.
↓ ↓ ↓ ↓
$[_S[_{DP}$ John's mother] $[_{VP}$ admires Mary]]

Notice that after the finite verb has been merged with the DP *John's mother*, no future operation can affect the internal structure of that DP. This follows from the assumption that merge is always to the right edge. All left branches become *phases* in the sense of (Chomsky 2000, 2001). This "left branch phase condition" was argued for specifically by (Brattico and Chesi 2019a), who use data from pied-piping to support it empirically. We can formulate the condition tentatively as (10), but a more rigorous formulation will be given as we proceed.

(10) *Left Branch Phase Condition (LBPC)*
Derive each left branch independently.

The intuitive content of this principle is that we "throw all left branches away" from the active working space as they are being assembled (11).

(11) John's + mother + admires + Mary
[John's mother] ←Throw away!
[ ] + admires + Mary (Already packaged: "John's mother")

An alternative formulation for (10) is 'XP is a phase if and only if it constitutes a left branch'. If no future operation is able to affect a left branch, as assumed in (10), then all grammatical operations (e.g. movement reconstruction) that must to be done in order to derive a complete phrase structure must be done to each left branch before they are sealed off by Merge. Furthermore, if, after all operations have been done, the left branch fragment remains ungrammatical or uninterpretable, then the original merge operation that created

the left branch phase must be cancelled. This limits the set of possible merge sites further. Any merge site that leads into an ungrammatical or uninterpretable left branch can be filtered out as either unusable or ranked lower.

2.4 Labeling

Suppose we reverse the arguments (9) and derive (12).

(12)

Mary	admires	John's	mother
↓	↓	↓	↓
Mary	[admires	[John's	mother]]
	[COMP:D]		

The verb *admires* selects for a DP-argument, but the complement selection feature itself only refers to the *label* of the complement phrase. The system must figure out the label of any complex constituent, such as *John's mother*. A recursive labeling algorithm (13) is postulated for this purpose. The intuitive function of the algorithm is to "search for the closest possible primitive head from the phrase," which will then constitute the label.

(13) Labeling

Suppose XP is a complex phrase. Then

a. if the left constituent of XP is primitive, it will be the label; otherwise,

b. if the right constituent of XP is primitive, it will be the label; otherwise,

c. if the right constituent is not an adjunct, apply (13) to it; otherwise,

d. apply (13) to the left constituent (whether adjunct or not).

The term *complex constituent* is defined as a constituent that has both the left and right constituent; if a constituent is not complex, it will be called *primitive constituent*. Notice that, according to this definition, a constituent that has only the left or right constituent will still be primitive (this becomes relevant later: it is like a head with a clitic on it). Conditions (13)c-d mean that labeling – and hence selection – ignores right adjuncts. This is in fact used in this study as a defining feature of adjuncts. If both the left and right constituents are adjuncts, the labeling algorithm will search the label from the left. This is a slightly anomalous situation, which we might consider ruling out completely. However, it leads into empirically correct results in some cases and is therefore preserved. Intuitively this means that if both the left and right constituents are adjuncts, then the left constituent is not an adjunct for the purposes of labeling and selection.

Consider again the derivation of (1), repeated here as (14).

(14) John + sleeps.

↓ ↓

[John, sleeps]

If *John* is a primitive constituent having no left or right daughters, as seems to be the case, the labeling algorithm will label [*John sleeps*] as a DP: its structure is [D V]. The primitive left constituent D will be its label. This is a problem. There are at least three ways to solve this problem. One solution is to reconsider the labeling algorithm. That seems implausible: (13) captures what looks to be a general property of language, thus this alternative would require us to treat (14) and other similar examples as exceptions. They are not exceptions, however: there is nothing anomalous about (14). The representation should come out as a VP, with the proper name constituting an argument of the VP of its head. Thus, the proper name should not constitute the (primitive) head of the phrase. There are two way to accomplish such outcome. One possibility is to redefine the notions of "primitive" and "complex" constituents in a way that would make proper names complex constituents despite having no constituents. If *John* were a complex constituent, then the labeling algorithm would take the (primitive) verb as the label. But the idea that *John* is a "complex constituent" despite having no constituents, left or right, suggest that we are confusing something. It is after all the case that *John* does not have any constituents here. A more plausible alternative would be to define the labeling algorithm in relation to another property that correlates with the distinction between primitive and complex constituents. We could say that *John* is a primitive constituent that has a special property of, say, "MAX" that makes is unable to project further; labeling would then respond to this property instead of phrase structural complexity. This is a theoretical possibility, but I find it unilluminating: it hides the real reason why *John* does not project while *sleep* does. In addition, the assumption is not innocent in the context of parsing, because it requires that the parser-grammar knows, of each element it processes, whether it is MAX or not MAX. If the property is mysterious, we must stipulate it for every constituent; or even worse, it could make every item ambiguous for the parser.

For these reasons, I suggest we keep the definition of labeling as provided and assumed that *John* is a complex constituent despite of appearing as if it were not. I assume that its structure is [D N], with the N raising to D to constitute one phonological word. This information can only come from the lexicon/morphological parser, in which proper names are decomposed into D + N structure. The structure of (14) is therefore (15), with "D" and "N" coming from the lexicon/morphology.

(15) John + sleeps.

↓ ↓

D N sleeps

↓ ↓ ↓

$[_{VP}[_{DP}D$ N] sleeps]

It is important to notice that the labeling algorithm (13) presupposes that primitive elements, when they occur in prioritized (i.e., left) positions, always constitute heads. A head at the right constitutes a head if there is a phrase at left. This happens in (15), which means that the whole phrase will come out as verb phrase. The outcome will be the same if the verb is transitive. The structure and label are provided in (16).

(16) $[_{VP}[_{DP}$ D N] [V^0 $[_{DP}$ D N]]

These are the correct properties.

2.5 Adjunct attachment

Adjuncts are geometrical constituents of the phrase structure, but they are stored in a secondary syntactic working memory and are invisible for sisterhood, labeling and selection in the main working memory. Thus, the labeling algorithm specified in Section 2.4 ignores adjuncts. The result is that the label of (17) is V: while analyzing the higher VP shell, the search algorithm does not enter inside the adverb phrase; the lower VP is penetrated instead (13)c-d.

(17) H^0 $[_{VP}$ John $[_{VP}[_{VP}$ admires Mary] $\langle_{AdvP}$ furiously$\rangle$]]

The reasoning applies automatically to selection by the higher head H: if H has a complement selection feature [COMP:V], it will be satisfied by (17). Consider (18).

(18) John [sleeps $\langle_{AdvP}$ furiously$\rangle$]

The adverb(ial) constitutes the sister of the verbal head V^0 and is potentially selected by it. This would often give wrong results. This unwanted outcome is prevented by defining the notion of sisterhood so that it ignores right adjuncts. The adverb *furiously* resides in a secondary syntactic working memory and is only loosely (geometrically) attached to the main structure; the main verb does not see it in its complement position at all. From the point of view of labeling, selection and sisterhood, the structure of (18) is $[_{VP}$[*John*] *sleeps*]. The reason adjuncts must constitute geometrical parts of the phrase structure is because they can be still targeted by several syntactic operations, such as movement and case assignment (Agree).

The fact that adjuncts are optional is explained by the fact that they are automatically excluded from selection and labelling: whether they are present or absent has no consequences for either of these dependencies. This explains also the fact that their number is not limited. On the other hand, these assumptions also entail that they can be merged anywhere, which is not correct. I assume that each adverbial (head) is associated with a feature *linking* or *associating* it with a label or feature. The linking relation is established by means of an 'inverse probe-goal relation' that I call *tail-head relation*. For example, a VP-adverbial is linked with V, a TP-adverbial is linked with T, a CP-adverbial is linked with C, and so on.

Linking is established by checking that the adverbial (i) occurs inside the corresponding projection (e.g., VP, TP, CP) or is (ii) c-commanded by a corresponding head (19).

(19) Condition on tail-head dependencies

A tail feature F of head H [TAIL:F] can be checked if either (a) or (b) holds:

a. H occurs inside a projection whose head has F.

b. H is c-commanded by a head whose head has F.

Of these conditions, (i) is uncontroversial. Condition (ii) allows adverbials to remain in their right-adjoined or extraposed positions in the canonical structure. If condition (ii) is removed from the model, then all adverbials will be reconstructed into positions in which they are inside the corresponding projections (reconstruction will be discussed in Section **Error! Reference source not found.**). A VP-adverbial will be reconstructed inside a VP, and so on. I will keep (ii); but the matter becomes important only when we address the 'free word order' property. A tail-head relation is checked by (19).

If an adverbial/head does not satisfy a tail feature, it will be reconstructed into a position in which it does. This operation will be discussed in Section 2.7.6. But the fact that constituents can satisfy some condition in a position different from the one inferred from the surface string means that most grammatical rules checking features must be specified in relation to some 'stage' in the derivation. An important property of the rule (19) and many similar rules is that each constituent must satisfy such rules only once, or in one position. Ignoring the ordering of operations in the derivation, if $C_1, \ldots C_n$ is a set of positions of a constituent in a chain formed by grammatical operation(s) such as movement, then the weakest possible criterion is that *one* member of the chain must satisfy the condition, but it does not matter which one. A stronger condition, the one that is relevant for (19), is that the last position must satisfy the condition. The intuitive motivation is that the condition is relevant at LF for the semantic interpretation, which must indeed pair adverbials with events expressed by verbs in order to arrive that proper semantic interpretation. In the parser-grammar framework, the representation that is generated last is the one that is also fed to LF. Thus, when the condition is checked at LF before the phrase structure is passed to systems responsible for extrasyntactic interpretation, the copies or traces (if any) after ignored and only the last position will be checked. This is not always the case, however. The EPP property (Section 2.6) of some head need not be checked by the last element in a chain: it can be the first, second, last, or any other member in between.

2.6 EPP

Some languages, such as Finnish and Icelandic, require that the specifier position of the finite tense is filled in by some phrase, but it does not matter what the label of that phrase is. This is captured by *unselective specifier features* [SPEC:*] and [!SPEC:*]. These features check that a phrase of any label (hence *) fills in the specifier position of the lexical element (SPEC:*), or requires it (!SPEC:*). Because the feature is unselective, it is not interpreted thematically, it cannot designate a canonical position, and hence the existence of this

feature on a head triggers A´/A movement reconstruction (Section 2.7.2). This constitutes a sufficient (but not necessary) feature for reconstruction; see 2.7.2. Language uses phrasal movement, and hence an unselective specifier feature, to represent a null head (such as C) at the PF-interface (Section 2.7.1). Corresponding to (SPEC:*) we also have (!COMP:*), which is the property that all functional heads have by definition.

2.7 Move

2.7.1 Why movement: entanglement between word order and morphosyntax

A moved *wh*-phrase represents the null head C(*wh*) at the PF-interface. A feature that cannot be represented at PF by means of a phonologically interpretable feature can be represented by moving a phrase to it vicinity (SPEC position), thus, to mark its existence at the PF-interface. This makes it possible to infer the existence of a grammatical head from word order. The opposite is true as well. In a free word order language such as Finnish, it is possible to infer word order from rich morphosyntax. The two-way relation represents an "entanglement" between order and morphology/phonology (Brattico 2019). We can think of the two phonological properties, order and morphosyntax, as being the two side of the same coin.

2.7.2 A´/A reconstruction

A phrase or word can occur in a canonical or noncanonical position. These notions get a slightly different interpretation within the comprehension framework. A canonical position *in the input* could be defined as one that leads the parser-grammar to merge the constituent directly into a position at which it must occur at the LF-interface, the latter which then constitutes a canonical position in terms of the finished phrase structure and semantic interpretation. For example, regular referential or quantificational arguments must occur inside the verb phrase at the LF-interface in order to get thematic roles. The problem for the parser-grammar is to determine when a phrase or a head in the input must be reconstructed into such canonical LF-position. There are at least two necessary conditions. One condition is the occurrence of a lexical element, such as the finite tense, that has the [SPEC:*] feature and has a phrase at its specifier position. The position is not thematic, hence the parser can infer that the phrase must be reconstructed to a canonical LF-position in which it is selected thematically. Another condition is the occurrence of a criterial feature. Criterial features are used to infer the existence of a licensing head. Both conditions are used in this study to trigger reconstruction. While the existence of criterial feature has an obvious functional motivation, mentioned above, the existence of the (SPEC:*) feature has remained a mystery. My view is that it represents a situation in which the "criterial feature" is constituted by a φ-set (Brattico 2016; Brattico and Chesi 2019b), but this assumption has not been fully implemented in this version.

2.7.3 A´-reconstruction and criterial features

Consider (20).

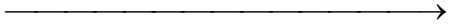
(20) Ketä Pekka ihailee __? (Finnish)
who.par Pekka admires
'Who does Pekka admire?'

The clause begins with two arguments, the first which hosts a criterial *wh*-feature. To compute properties of (20) correctly, the parser-grammar must infer the existence of a operator head C(*wh*) on the basis of the moved phrase and the criterial feature (21). This is, in fact, the function of movement: to signal the force and scope of the interrogative clause.

(21) Ketä C(wh) Pekka ihailee __?
wh C(wh)
who.par Pekka admires
'Who does Pekka admire?'

The mechanism has two computational steps that must be distinguished. The first is the recognition that a head is missing. That can be inferred (again, in this particular example) from the existence of what looks to be the two specifiers of the T/fin head. (If the head is not missing, as in the case of English *did* support, nothing needs to be done.) The parser-grammar will *generate* the required head to the structure. The head will be generated into a position in which the phrase will be its unambiguous specifier, as shown in (21). The next step is to generate or infer the label of the head. This information is obtained from the criterial feature. Thus, if the moved phrase contains a *wh*-feature, that feature will be 'copied' from the phrase to the head, as shown in (21). If the phrase had another feature, such as contrastive focus, then that feature would be copied. This allows the parser-grammar to infer both the existence and the nature of the phonologically null head. This 'copying' operation resembles Agree of the standard bottom up theory; I will discuss it later in detail.

The phrase must be reconstructed back to its canonical LF-position. The intuition is that it will be stored into a syntactic memory buffer and reconstructed into an empty position downstream. The mechanism was first proposed and developed in (Chesi 2004) in connection with the top-down theory. To implement the system formally, we need to specific two further facts: how are empty positions detected and when does reconstruction occur? When it comes to the former problem, I will adopt Chesi's original idea that empty positions are determined by lexical selection features. If a head is encountered that selects for a label L, but the selected constituent is not present, memory buffer is consulted to determine if a suitable phrase is found; if it is, the phrase will be merged from the memory buffer to the empty position. In the example (22), the interrogative pronoun will be first stored into the memory buffer and then reconstructed into the complement of the transitive verb once the parser-grammar detects that a required argument is missing (i.e. its [!COMP:D] feature is not satisfied).

(22) Who does John admire __?


I will assume, for reasons that become clear later, that deconstruction is a form of copying. Thus, the interrogative pronoun is copied from the criterial position into the memory buffer, from where it is copied to the canonical LF-position designated by "__".

Chesi assumes that phrases are stored into the memory buffer and retrieved into canonical LF-positions in tandem by consuming words. This is possible only if the intended phrase structure is known in advance, as it is in Chesi's approach; it is not possible in the context of parsing, in which we are working with PF-interface information (linear string of words). I will assume that movement is reconstructed separately for each left phrase (phase) (Brattico and Chesi 2019a). Formally, movement is reconstructed inside XP during Merge(XP, α). This condition is captured by (23).

(23) Merge(α, β) ↔ Reconstruct(α)

There are several reasons why this assumption is necessary. The most obvious reason is that it is impossible to assess the grammaticality of XP unless we attempt to reconstruct movement inside XP. Leaving movement undone would leave several thematic positions empty, and the construction would evaluate as ungrammatical/uninterpretable. There is only one situation in which (23) is not valid: when the final phrase structure is submitted to the LF-interface. During that process, reconstruction is applied to the top node without Merge (unless, of course, we assume that even during the final step something is merged to the right of the final phrase structure).

2.7.4 A-reconstruction and EPP

Another sufficient condition for phrasal movement is the occurrence of a phrase at the specifier position of a head that has the [SPEC:*] feature (=EPP in the standard theory). The mechanism is the same as that of the A´-reconstruction (Section 2.7.3), but the grammatical features involved differ. The 'criterial features' of this operation are the φ-features and not, say, *wh*-features, as shown in (24).

(24) Me ihaile-mme Merjaa. (Finnish)
1pl.nom admire-**1pl** Merja.par
'We admired Merja.par.'

The finite tense node agrees in φ-features with a phrase that typically (but not necessarily) moves to its specifier position. The copying/agreeing operation is the same as it was in the case of criterial features: features from the moved phrase are copied to the head (24).

2.7.5 Head reconstruction

Many heads occur in noncanonical positions in the input string. In Finnish, for example, verb-initial clauses are ungrammatical due to the [SPEC:*] feature at T/fin, unless a head has been moved to the C-head either to generate a verum focus interpretation (corrective focus scoping over the whole sentence) or to create some

other interpretation associated with a criterial feature present in the head. In the example (25), a head in an embedded finite clause has been suffixed with the yes/no clitic -kO and then fronted.

(25) Nukkua-ko Pekka ajatteli että hänen pitää _?
sleep-Q Pekka thought that he must
'Was it sleeping that Pekka thought that he must do?'

Lexical and morphological parser first provides the parser-grammar with the information that the -kO particle in the first word encodes the C-morpheme itself (C(-kO), which is then fed into the parser-grammar together with the rest of the morphological decomposition of the head. In this case, the verb *nukkua-ko* is composed out of C(-*kO*), infinitival T_{in} (-*a*-) and V (*nukku*-). Morphology extracts this information from the phonological word and feeds it to syntax (26). Symbol "#" indicates that there is no word boundary between the morphemes/features.

(26) C(-kO) + #T_{inf} + #V + Pekka + ajatteli + että + hänen + pitää
C T V Pekka thought that he must

Morphology has no access to syntax; instead, it decomposes phonological words and feeds them into syntax in a linear order, one morpheme at a time. The individual heads are then collected together in syntax into one complex head. The incoming morphemes are "stored" into the right constituent of the first morpheme, a process that resembles cliticalization. Thus, if syntax receives a word-internal morpheme, it will be merged to the right edge of the previous morpheme: [$_{\alpha}$ ∅ β]. I will denote it as α{β}. Notice that by defining "complex constituent" as one that has both the left and right constituent, [$_{\alpha}$ ∅ β] comes out as "primitive constituent" and can therefore constituent the head of a projection. The linear sequence C-T-V becomes C{T{V}}(or [$_{C0}$ ∅ [$_{T0}$ ∅ V0]]), and this is what gets merged (27).

(27) C{T_{inf}{V}} Pekka ajatteli että hänen pitää __
Pekka thought that he must

Head reconstruction will return the constituents of the complex head into their canonical positions when the whole phrase is merged as a left constituent. This is done by targeting the highest head inside a given complex head (i.e. T_{inf}{V} is targeted inside C) and finding the closest position in which it can be selected, and in which it does not violate local selection rules. In the example (27), this will be the T_{inf}{V}. The closest possible position for T_{inf} is the empty position inside the embedded clause. V will be extracted from the complex head in the same way and reconstructed into the complement position of T_{inf}. The same operation will extract D and N from a proper name such as *John*: D{N} → [D N].

(28) C{T_{inf}{V}} Pekka ajatteli että hänen [pitää [T_{inf}{V} V]].
⟶ ⟶

This operation reverse-engineers head movement.

2.7.6 Adjunct reconstruction

Consider the pair of expressions in (29) and their canonical derivations.

(29)

a. Pekka käski meidän ihailla Merjaa.
Pekka.nom asked we.gen to.admire Merja.par
↓ ↓ ↓ ↓ ↓
[Pekka [asked [we [to.admire Merja]]]]
'Pekka asked us to admire Merja.'

b. Merjaa käski meidän ihailla Pekka.
Merja.par asked we.gen to.admire Pekka.nom
↓ ↓ ↓ ↓ ↓
[Merja [asked [we [to.admire Pekka]]]
'Pekka asked us to admire Merja.'

Derivation (a) is correct, (b) is incorrect. This is so because the thematic roles are identical in both examples. Neither A´- nor A-reconstruction can handle these cases. The problem is created by the grammatical subject *Pekka* 'Pekka.nom', which has to move 'upwards' in order to reach the canonical LF-position Spec,VP. The system introduced so far has no operation that achieves this. Because the distribution of thematic arguments in Finnish is very similar to the distribution of adverbials, I have argued that richly case marked thematic arguments can be promoted into adjuncts (Brattico 2016, 2018). See (Baker 1996; Chomsky 1995: 4.7.3; Jelinek 1984) for similar hypothesis. This can be captured in the following way (from (Brattico 2019)). Suppose that case features must establish local tail-head (inverse probe-goal) relations with functional heads as provided in (30).

(30) Case features must establish local tail-head relations such that
a. [nom] is checked by +FIN,
b. [acc] is checked by v (ASP),
c. [gen] is checked by −FIN
d. [par] is checked by −PHI.

The symbol "−PHI" refers to a head that never exhibits φ-agreement. If the condition is not checked by the position of an argument in the input, then the argument is treated as a an adjunct and reconstruction into a position in which (30) is satisfied. In this way, the inversed subject and object can find their ways to the canonical LF-positions (31). Notice that because the grammatical subject *Pekka* is promoted into adjunct, it no longer constitutes the complement of the verb; the partitive-marked direct object does.

(31) [⟨Merjaa⟩$_2$ T/fin [__$_1$ [käski [meidän [ihailla [__$_2$ ⟨Pekka⟩$_1$]]]]
+FIN ⟷ [nom] −PHI ⟷ [par]
Merja.par asked we.gen to.admire Pekka
'Pekka asked us to admire Merja.'

Rich case suffixes are 'adverbializers': they allow an argument to find its way into a canonical LF-position irrespective of its position in the word order. Recall that the same mechanism reconstructs also genuine adverbials: a VP-adverbial will find its way into the vicinity of the V-head, a TP-adverbial into the vicinity of T, and so on.

2.8 Ordering of operations

The catalog of computational operations required to construct a phrase structure from a linear string of words must itself be ordered in some way. There seems to be only one way to order them. Both A´/A-reconstruction and adjunct reconstruction presuppose head reconstruction, because it is only the presence of heads and their lexical features that can guide A´/A- and adjunct reconstruction. The former relies on EPP features and empty positions, whereas the latter relies on the presence of functional heads. Furthermore A´/A-reconstruction relies on adjunct reconstruction: empty positions cannot be recognized as such unless orphan constituents that might be hiding somewhere are first returned to their canonical positions. All three movement operations presuppose merge from the input, and the whole sequence presuppose that the merge has constructed a left branch. The whole sequence must therefore be (32).

(32) Derive $\alpha \rightarrow$ Merge [α, β] $\rightarrow$ head reconstruct $\alpha \rightarrow$ adjunct reconstruct $\alpha \rightarrow$ A`/A reconstruct α

2.9 Lexicon and morphology

Most phonological words enter the system a polymorphemic units that might be further associated with inflectional features. The morphological component is responsible for decomposing phonological words into these components. A table-look up dictionary matches phonological words directly with their morphological decompositions. This corresponds with an automatized pattern recognition procedure. The decomposition consists of a linear string of morphemes m_1#...#m_n that are separated and inserted into the linear input individually (33). Notice the reversed order.

(33) Nukku-u-ko + Pekka → Q + T/fin + V + Pekka
sleep-T/fin-Q Pekka ↓ ↓ ↓ ↓
sleep#T/fin#Q [Q [T/fin [V Pekka]]]

The primitive morphemes are matched with the lexicon (table-lookup) and retrieve lexical items. Lexical items are provided to the syntax as primitive constituents, with all their properties (features) coming from the

lexicon. Another morphological system is comprised of generative morphology that is called for when an input word does not match with anything in the table-lookup dictionary. Generative morphology parses unrecognized words and/or guesses their feature composition (e.g., label).

Inflectional features (such as case suffixes) are listed in the lexicon as items that have no morphemic content. They are extracted like morphemes, inserted into the input sequence, but converted into *features* instead of morphemes in syntax. An inflectional feature F in a sequence m_1#...m_i#F will become a feature of the preceding morpheme m_i: *Merja-a* 'Merja.par' = N#D#par = [D(par, def…) N].

Lexical features emerge from three distinct sources. One source is the language-specific lexicon, which stores information that is specific to a particular lexical item in a particular language. For example, the Finnish sentential negation behaves like an auxiliary, agrees in φ-features, and occurs above the finite tense node in Finnish (Holmberg et al. 1993). Its properties differ from the English negation *not*. Some of these properties are so idiosyncratic that they must be part of the language-specific lexicon. One such property could be the fact that the negation selects T as a complement, which must be stated in the language-specific lexicon to prevent the same rule from applying to the English *not*. It is assumed that language-specific features *override* features emerging from the two remaining sources if there is a conflict.

Another source of lexical features comes from a set of universal redundancy rules. For example, the fact that the small verb v selects for V need not be listed separately in connection with each transitive verb. This fact emerges from a list of universal redundancy rules which are stored as feature implications. These redundancy rules make up a sort of 'mini grammar'. In this case, the redundancy rule states that the feature [CAT:v] implies the existence of feature [COMP:V]. Broad verb classes (e.g., transitive, unaccusative) can be defined as (macro)features that are associated with a set of redundancy rules. When a lexical item is retrieved, its feature content is first fetched from the language specific lexicon and is then processed through the redundancy rules and parameters. If there is a conflict, the language-specific lexicon wins.[1] Lexical features constitute an unstructured set, but the features themselves can have a 'type:value' structure. For example, labels are provides as values (N, V, A, …) of the type CAT(egory), so the actual features are of the form (CAT:L).

[1] A third source of lexical features comes from a curious set of parametric rules. For example, in Finnish most heads that have the [SPEC:*] feature also exhibit φ-agreement in some context (e.g., prepositions, noun heads), whereas the same functional heads in English have neither (Brattico and Chesi 2019b). Such generalizations could be stored in the language-specific lexicon, but this would be redundant to the extent that a general rule is at stake. Because the rule is still language-specific, it cannot be part of the list of universal redundancy rules. It is unclear what the source and implementation of these rules is; they look like language-specific parts (or parametrizations) of the mini-grammar defined by the redundancy rules, but it is unclear what the term 'language-specific' really means and what it means to have 'parameters'. These regularities are, therefore, stipulated in a third component of the lexicon.

2.10 Argument structure

The term “argument structure” refers to the structure of thematic arguments at their canonical LF-positions, the latter which are defined both by means of theta role assignment and by tail-head dependencies. The parser-grammar will usually have to reconstruct the argument structure from the input string due to several displacement operations.

The thematic role of ‘agent’ is assigned at LF by the small verb v to its specifier. The small verb therefore has feature [!SPEC:D]. A DP argument that occurs at this position will automatically receive the thematic role of ‘agent’. The parser-grammar does not ‘see’ the interpretation, only the selection feature. Thus, when examining the output of the parser, the canonical positioning of the arguments must be checked against native speaker interpretation. The sister of V receives several roles depending on the context. In a v-V structure, it will constitute the ‘patient’. In the case of an intransitive verb, we may want to distinguish left and right sisters: right sister getting the role of patient (unaccusatives), left sister the role of ‘agent’ (unergatives). Their formal difference is such that a phrasal left sister of a primitive head constitutes both a complement and a specifier (per formal definition of ‘specifier’ and ‘complement’), whereas a right sister can only constitute a complement. This means that unaccusatives and unergative verbs can be distinguished by means of lexical selection features: the latter, but not the former, can have an extra specifier selection feature, correlating with the ‘agentive’ interpretation of the argument. Thus, a transitive verb will project three argument positions Spec,vP, Spec,VP and Comp,VP, whereas an intransitive two, Spec,VP and Comp,VP. This means that both constructions have room for one extra (non-DP) argument, which can be filled in by the PP. Ditransitive clauses are built from transitive template by adding a third (non-DP) argument. They can be selected, e.g. the root verb component V of a ditransitive verb can contains a [!SPEC:P] feature. Ideally, verbal lexical entries should contain a label for a verb class, and that feature should be associated with its feature structure by lexical redundancy rules.

Adverbial and other adjuncts are associated with the event by means of tail-head relations. A VP-adverbial, for example, must establish a tail-head relation with a V. Adverbial-adjunct PPs behave in the same way. The tail-head relation can involve several features. For example, the Finnish allative case (corresponding to English ‘to’ or ‘for’) must be linked with verbs which describe ‘directional’ events (34). It therefore tails a feature pair CAT:V, SEM:DIRECTIONAL. There is no limit on the number of features that a verb can possess and a prepositional argument that tail.

(34)

a.	*Pekka	näki	Merjalle.
	Pekka	saw	to.Merja
b.	Pekka	huusi	Merjalle
	Pekka	yelled	at.Merja

3 Formalization and implementation

3.1 Introduction

The framework delineated in Section 2 consists of assumptions that follow, some of them by virtual necessity, from the assumption that parsing is incremental and uses the core computational operations such as Merge. A full formalization and computational implementation, on the other hand, requires a set of conjectures, working hypotheses and further details that can be demonstrated to be "functional" by means of computer simulation but are less inevitable and thus subject to alternative formalizations and approaches. In addition, there are several ideas that are part of the overall framework but not implemented, such as various operation that have to do with discourse semantics and communicative pragmatics.

3.2 Overall structure of the algorithm

The parser-grammar implements a recursion over the whole search space defined by the ranking of the merge sites. The basic structure of the recursion is illustrated in Figure 1.

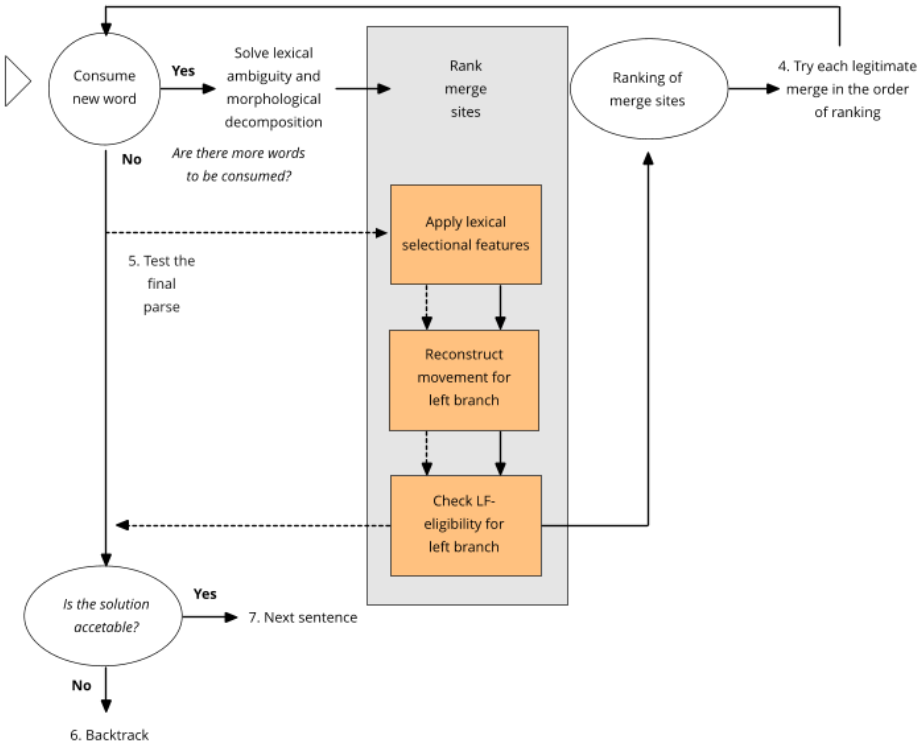


Figure 1. The structure of the language comprehension algorithm.

The individual components are commented in the subsections below.

3.3 The parser

3.3.1 Phillips cycle

The recursive parsing function (__first_pass_parse) takes the currently constructed phrase structure XP, a list of words and an index in the list of words as its arguments. It will first check if the whole clause has been consumed and, if it is, XP will be reconstructed (Section 3.4.2), tested for LF-legibility (Section 3.8) and transferred to LF. Reconstruction attempts to undo movement, LF-legibility tests that the output is semantically interpretable, while the LF-transfer function is in the current implementation the locus of binding theory and related operations. If it passes all phases, the result will be accepted: the system has "understood" the sentence. If the algorithm is set to find only one (first) solution, parsing will terminate. If the algorithm is set to find all solutions, it will backtrack. The latter option will produce all ambiguities (if they exist).

Suppose word w was consumed. To retrieve properties of w, the lexicon will be accessed (access_lexicon) with w as an argument. This operation corresponds to a stage in real language comprehension in which an incoming sensory-based item is matched with a lexical entry. If w is ambiguous, all corresponding lexical items will be returned as a list [w1, w2, . . ., wn] that will be explored in some order (added to the recursion). The ordering is arbitrary in the current interpretation but not so in a realistic parsing scenario, hence in a future version the system should include a separate function that orders the list of lexical items matching any given phonological string or substring, on the basis of the preceding phrase structure, discourse context and encyclopedic knowledge.

Next, a word from this list, say w1, will be subjected to morphological parsing (morphological_parse, Section 3.6). Morphological parser will return a new list of words that contains the individual morphemes that were part of w1 together with the lexical item corresponding with the first item in the new list. The new list contains a morphological decomposition of the word, if any. Some of the morphemes in word w could be inflectional: they are stored as features into a separate memory buffer and then added to the next morpheme when it is being consumed. If inflectional features were encountered instead of a morpheme, the parsing function is called again immediately without any rank and merge operations. If there were several inflectional affixes, they would be all added to the next morpheme m consumed from the input (35).

(35) Inflectional feature F1 + F2 + F3 + m (Input string)
→ m{F1, F2, F3} (Result)

Suppose a morpheme m was consumed. The morpheme must now be merged to the existing phrase structure into some position. First, merge sites that can be ruled out as impossible by using local information are filtered out (filter). The remaining sites are then ranked (ranking)(Section 3.3.2). Each site from the ranking

is then explored. For each site there are two options: if the new morpheme α was part of the same word as the previous morpheme β, it will be "sink" into the word to create a complex head β{α}. If not, α will be merged countercyclically to the existing phrase structure, and the parsing function is then called recursively. If a ranked list has been exhausted, the algorithm will backtrack to an earlier step.

3.3.2 Filter and ranking

Filtering. Several conditions can be exploited in filtering potential merge sites so that they do not appear in the rankings and do not consume computational resources. (1) If the new morpheme was inside the same word as the previous one, all other solutions expect merge to the complement will be filtered out. (2) Primitive sites that do not accept any type of complements (–COMP:*) are filtered out. (3) If a site is not primitive, it will be reconstructed silently (Section 3.4.2) and tested for LF-legibility. If the new morpheme is adjoinable (and could become right adjunct), then we do not know if αP will be left branch in the final output and we abandon any attempts at rejection; otherwise, LF-legibility can be used. (4) In addition, failure to satisfy the probe-goal test, head integrity test or the criterial feature integrity test (Section 3.8) will lead into permanent rejection: these tests cannot be satisfied by any further operation. There could exist other criteria for filtering and rejection. On the other hand, these tests must be used conservatively, as any site rejected here will never be considered, and the whole parsing branch will be closed permanently.

Ranking. The ranking function gives each solution a numerical value based on various criteria, with larger number indicating priority. If two sites have the same ranking, lower sites in the phrase structure will be prioritized and therefore explored first. This rule affects computational complexity significantly, but it has no independent theoretical motivation; it can be regarded as a heuristic. The ordering does not affect solutions found by the algorithm, only efficiency and the first solution returned by the algorithm. Their justification is in their ability (or inability) to guide the parser efficiently to a solution. Suppose the new element is h and the site to be tested is S. The criteria used in the current version are the following:

(36) *Ranking criteria*

a. Positive specifier selection (+): S matches for h's specifier selection,
b. Negative specifier selection (+): S matches for h's negative specifier selection,
c. Negative specifier selection of everything (–): h has [–SPEC:*],
d. Infrequent specifier feature (–): S matches for h's infrequent specifier selection (not used),
e. Do not break existing head dependencies (–): [S h] would break existing α^0-β^0 dependency,
f. Tail-head test (–): check if [S h] would fail h's tail features,
g. Complement test (+): S (or any morpheme inside S) selects for h as complement,
h. Negative complement test (–): S (or any morpheme inside S) does not select for h as complement,
j. Semantic mismatch (–): S and h's semantic features mismatch,
k. Left branch evaluation (–): Reconstructed S fails LF-legibility,

l. Adverbial tail-head test (–): h has label Adv but fails tail-head test when c-commanded by T/fin,

m. Adverbial tail-head test (+): h has label Adv and satisfies tail-head test when c-commanded by T/fin.

If all solutions are ranked negatively, a "geometrical" solution will be adopted, according which the largest S will be prioritized that is adjoinable and does not contain T/fin, the latter which means that we do not try to merge above T/fin. This heuristic rule works in practice and has no theoretical or intuitive motivation.

The ranking system adopted here has produced extremely efficient solutions for the empirical phenomena explored thus far. It is expected, however, that many more factors affect ranking, and that the above list is incomplete and partially wrong. These insufficiencies will emerge later when more empirical phenomena are explored.

3.4 Definitions for formal primitives

3.4.1 Merge: cyclic and countercyclic

The simplest form of Merge takes two constituents α and β and joins them together to form [α, β], α being the left constituent, β the right constituent. Countercyclic Merge is a more complex operation. It targets a constituent α inside an existing phrase structure and creates a new constituent γ by merging β either to the left or right of α: γ = [α, β] or [β, α]. Constituent γ then replaces α in the phrase structure, with the phrase structural relations updated accordingly. If α is primitive and an adjunct, γ will be adjunct as well (in other words, if the operation takes place in the "secondary syntactic working space," the resulting complex constituent will also be there). Countercyclic Merge is not symmetric: β is attached to a node α inside a complex phrase structure, not vice versa. An inverse operation of countercyclic Merge is remove (remove), which removes a constituent from the phrase structure and repairs the empty hole left behind. This operation is used when the parser attempts to reconstruct movement: it merges elements into candidate positions and removes them if they do not satisfy a given set of criteria.

3.4.2 Sister, specifier, complement

Iterative application of merge creates a structural entity that is used to define certain basic relations. Fundamentally these relations will be used by the conceptual-intentional system to recover the meaning of the expression; inside the parser-grammar, they appear as formal relations. A *sister* of α is the non-α daughter of the first branching node from α that is not right-adjunct. In addition, right adjuncts don't have sisters, so that they are "isolated." It follows that in [α, β] both α and β are each other's sisters. In [⟨α⟩, β] the same holds, but in [α [β, ⟨γ⟩]], the sister of β is α. The intuitive content of this rule is that right adjuncts, here ⟨γ⟩, are in a separate syntactic working memory and are invisible for sisterhood. A separate relation of *geometrical sisterhood* takes them into account. A *specifier* of a left head α is the complex left sister of α's mother that is inside αP (based on labeling), and the specifier of a right head β is its complex left sister (i.e. XP in [XP, β]). Both left adjuncts and left non-adjuncts can be specifiers. A *complement* of α is its *right*

sister. LF-legibility uses a broader criterion, in which the complement selection feature is checked by sisterhood; the semantic interpretation is one degree more flexible than the formal system. The latter assumption (complement = sister) is the more correct one, thus it is possible that a later iteration of the model should abandon the former and use only the latter.

3.4.3 Downstream, upstream, left branch and right branch

Several operations require that the phrase structure is explored in a pre-determined order. Movement reconstruction, for example, my explore the phrase-structure in a determined order in order to find reconstruction sites. In a typical scenario, the right edge is explored: left branches are phases and not revisited (thus, movement also from within left branches is illicit). The operation of right edge exploration would be trivial to define were there no right-adjuncts, but because right adjuncts are possible, the definition of "right edge" requires a small clarification. The clarification is that they are ignored when we explore the right edge (for other purposes than merging new words to the phrase structure, see below). Thus, the right edge of [α, β] is β, but the right edge of [α, ⟨β⟩] is α. The intuitive motivation for this definition is that β is inside another syntactic working space. A *downstream* or *downward* relation follows this definition. The *left branch* of [α, β] will always be α. These definitions mean that for any given constituent there are three possible "directions": to go downstream (thus, follow labeling and selection), to turn right (to enter into a right adjunct), or to turn left (enter the left branch). The downward/downstream relations will be used in all movement reconstruction: thus, movement reconstruction into left branches or right branches will not occur, which are the exact correct empirical properties. Notice that adverbials are not A´-movement islands if they occur in the downward path, again the correct result empirically.

When the parser-grammar is consuming incoming words, the situation is requires a comment. Consider sentence (37).

(37) John	+	travelled	+	by	+	using	+	. . .		
↓		↓		↓		↓				
[John	[	travelled	[	by		using				

Let's make the untrivial assumption that the parser-grammar has already at this point determined that it is building an adverbial-adjunct *by using…* If so, then the above definition of "right edge" would require all incoming words to be merged as if the adjunct-adverbial were not present at all. The adverbial would be invisible and not part of the right edge. This gives the wrong result. The next word(s), say *his car*, must be merged inside the adverbial. On the other hand, it is not clear that the parser grammar can determine at this stage that it is constructing an adverbial. Another possibility is that it is constructing a complement or a PP argument for the main verb. Therefore, the phrase *by using* is part of the right edge insofar as it has not been promoted into an adjunct status, which is the case here.

3.4.4 Probe-goal

The probe-goal relation is used in the standard theory to check the existence of a feature (Chomsky 2000, 2001, 2008). It differs from complement specifier selection features in that whereas complement selection is local, probe-goal is not: the probe-goal is like long-distance complement selection. It is used in exactly this way in the present implementation. For example, consider the grammatical law which says that a finite C must always select for a finite T. We could define this as a simple selection feature (!COMP:T/FIN), but the strategy runs into troubles in Finnish in which the sentential negation head can intervene C and T(28).

(38) C Pekka ei nuku.
C Pekka Neg T
'Pekka does not sleep.'

The problem cannot be solved by assuming Finnish-specific selection C→Neg and Neg→T, because the negation is not obligatory, and adding the rule C→T would still allow the parser to create illegitimate C-H configurations, H being any head. What we need is a rule that *forces* the C→T connection, but if we posit it as obligatory complement selection, then (38) becomes ungrammatical because here C selects for Neg. The solution to this problem is obvious: there must exist an obligatory C→T selection that is nonlocal. This is achieved by associating C with lexical feature (!PROBE:CAT:T/FIN). The existence of the probe-feature triggers search for the *goal feature*, here (CAT:T/FIN)(probe). Suppose P is the probe head, G is the goal feature, and α is its (non-adjunct) sister in configuration [P, α]; then:

(39) Probe-goal
Under [P, α], G the goal feature, search for G from left constituents by going downwards inside α along its right edge (ignoring right adjuncts and left branches).

For everything else than criterial features, G must be found from a primitive head to the left of the right edge node. Thus, if [H, α] is at the right edge and H is a primitive constituent, feature G is searched from H. If H is complex, it will not be explored (unless G is a criterial feature). Probe-goal dependencies are subject to limitations that have not been explored fully. The present implementation has an intervention clause which blocks further search if a primitive constituent is encountered at left that has the same label as the probe, but the matter must be explored in a detailed study and is here implemented only provisionally. But consider against the C→T case. It is obvious that when C searches for T, it cannot satisfy the probe-feature by going through another (embedded) C. If it did, the lower T would be paired with two C-heads; this is semantically gibberish. Therefore, there is a "functional motivation" for intervention.

3.4.5 Tail-head relation

A tail-head relation is triggered by a lexical feature (TAIL:F_1,…,F_N), F… being the feature or set of features that is being searched from a head. In order for F to be "visible" for the constituent containing the tail-

feature, say T, F must occur in a primitive left head H at the upward path from T. An "upward path" is the path that follows the unambiguous mother-of relation. In the example (40), the tail-feature (TAIL:F) at T^0 can see the feature at the head A^0, and B^0, but not at X^0.

(40) $[_{AP}$ A^0 $[_{BP}[_{XP}$YP X] $[B^0$ $[T^0$ ZP]]]]
{F} ←——————— {TAIL:F}

For the tail-head relation to be satisfied, all tail-features (if there are several) must be satisfied by the one and the same head: *partial* feature match results in failure. Thus, if T has a tail feature (TAIL:F,G), but A only has the feature F, then the tail-head relation fails because there is only a partial match; consequently, A must possess both features F and G. For the tail-head relation, it does not matter if T is inside and adjunct or not.

3.4.6 Minimal finite edge

The *minimal finite edge* (local_minimal_tense_edge) in relation to a constituent α is the closest node X upstream from the constituent α (thus containing α) such as (i) X is inside a projection from label FIN (FinP) but (ii) X's sister is not inside a projection from label FIN (FinP). For example, in a configuration C-FinP, the minimal finite edge for any constituent inside FinP that is no contained in another finite projection is that FinP. The same procedure could be generalized so that it finds the edge of any projection.

3.5 Movement reconstruction

3.5.1 Introduction

Movement reconstruction is a 'reflex-like' operation that is applied to a phrase structure α and takes place without interruption from the beginning to the end without the possibility of backtracking. From an external point of view, it therefore constitutes 'one' step; internally the operation consists of a determined sequence of steps. All movement inside α is implemented if and only if Merge(α, β)(Section 2.3).

The operation, call it Reconstruct (reconstruct), targets a phrase structure α and follows a predetermined order: head movement reconstruction → adjunct movement reconstruction → A´/A-movement reconstruction. Head movement reconstruction can take place in one of two ways: (a) [α{β}, γ] → [[α, β], γ] or [α [$_{\gamma}$…β…]]. Option (a) will be selected if α is primitive, has an affix, and [α, β] satisfies LF-legibility (Section 3.8). In that case no further reconstruction operation will be applied to α; otherwise (α is complex or [α, b] does not satisfy LF-legibility) option (b) is selected (Section 3.5.2), followed by adjunct movement reconstruction (Section 3.5.3) and A´/A-movement reconstruction (Section 3.5.5). As said earlier, reconstruction is not an operation in which various alternatives are tried in some order; it is an all-or-none phenomenon.

3.5.2 Head movement reconstruction

Head movement reconstruction (reconstruct_head_movement) travels downward on the right edge of α, targets primitive constituents at the left that have an affix β{φ} and drops the head/affix φ downstream if a suitable position is found. Dropping (drop_head) is implemented by trying to "fit" the head to the left of each node at the right edge (drop_condition_for_heads). The position is accepted as soon as one of these conditions is met: (i) the head φ has no EPP feature and can be selected in that position by a higher head; (ii) it has an EPP feature (Section 3.7), has a specifier, and can be selected in that position by a higher head. The left complex sister constitutes an acceptable specifier [XP, φ]. These are the basic conditions which capture the notion that heads are reconstructed "as soon as a potential position is found." There are two exceptional conditions. One is if the bottom of the phrase structure has been reached by a head that has EPP and is selected properly. If the head has an EPP feature, is selected property, has no specifier, but occurs in configuration [α, β], β being a primitive head without affixes, the solution is accepted without a specifier, because β is a bottom node. The configuration is also accepted if the next left constituent downstream is primitive and cannot constitute a specifier (but selection applies). In the latter two cases, the expression can be saved later if the SPEC position is filled in by phrasal movement reconstruction; if not, the derivation will eventually crash. Head φ can itself be complex. The algorithm will encounter the complex φ{γ} later when it travels downwards from the position it originally extracted φ, and applies the same algorithm to γ. A consequence of these rules is that heads must be reconstructed into the first position in which it can be selected; lower positions are never explored.

If head reconstruction reaches the bottom, it will try to reconstruct the head to the complement of the bottom node. If no legitimate position was found but the search reaches the bottom, then the head will be reconstructed locally to [α [φ XP]] as a last resort; an unreconstructed head crashes at LF-legibility.

3.5.3 Adjunct reconstruction

Adjunct reconstruction (reconstruct_floaters) begins from the top of the phrase structure α and targets (detect_floater) floater phrases at the left and to the right (e.g., DP at the bottom). If a floater is detected, it will be dropped (drop_floater). A floater has the following necessary properties: (i) it is complex; (ii) it has not been moved already; (iii) it has a tail set; (iv) floating is not prevented by feature [−FLOAT]. Condition (iv) is stipulative and prevents genitive arguments from floating in Finnish, but this may also apply to English accusative pronouns. An alternative is to exclude these arguments from the tail-head mechanism, but then their canonical position must be retrieved without the tail features.

Suppose α satisfies conditions (i-iv) above. Then if the tail-head features (Section 3.4.5) are not satisfied in this position (external_tail_head_test, Section 3.5.4), α must be subject to adjunct reconstructions and is designated as a floater. The functional motivation of this condition is clear: we know that the constituent is not at its canonical position. In addition, (a) if its tail-head features are satisfied but α constitutes a specifier

of a finite head with an EPP feature, or (b) if it constitutes a specifier for a head that does not accept specifiers at all [−SPEC:*], it will also be targeted for reconstruction. Condition (b) is self-evident: the position is still wrong. Condition (a) is required when the adverbial and/or another type of floater occurs in the specifier of a finite head where its tail features are (wrongly) satisfied by something in the selecting clause. The EPP-feature indicates that the adverbial/floater must reconstruct into its own clause, and not to remain in the high specifier position.

After a floater is detected, it will be dropped (drop_floater). Dropping is implemented by first locating the closest finite tense node (locate_minimal_tense_edge). Starting from that and moving downstream, the floater is "fitted" into each possible position. Adverbials and PPs are fitted to the right, everything else to left. Fitting (is_drop_position) involves three conditions: (i) tail-head features must be satisfied (external_tail_head_test, Section 3.5.4); (ii) we are not trying to drop into the same projection where we started; (iii) we are not creating new specifiers for a head. Adverbials and PPs will be merged to right, they have to observe only (i); everything else to left, and by (i-iii). The floater is promoted into an adjunct (create_adjunct) once a suitable position is found. This will allow it to be treated correctly by selection, labeling and so on.

3.5.4 External tail-head test

External tail-head test (external_tail_head_test) checks if the tail-features of a head (i) can be checked by any of the c-commanding head(s) or if (ii) it is located inside the projection of such head. Tail features are checked in sets: if a c-commanding (i) or containing (ii) head checks all features of such a set, the result of the test is positive; if matching is only partial, negative. If the tail-features are not matched by anything, the test result is also negative. Thus, the test ensures that constituents that are "linked" at LF with a head of certain type, as determined by the tail features, can be so linked. All c-commanding heads are analyzed; this implements the feature vector system of (Salo 2003)(get_feature_vector). There are several reasons why checking must be nonlocal, long-distance structural case assignment being one of them.

3.5.5 A´/A-reconstruction

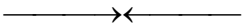
Suppose A´/A-reconstruction (reconstruct_phrasal_movement) is applied to α. The operation begins from the top of α and searches downstream for primitive heads at the left or (bottom) right. Three conditions are checked: (1) If the head h lacks a specifier it ought to have on the basis of its lexical features (e.g. v), memory buffer is searched for a suitable constituent and, if found, is merged to the SPEC position (fill_spec_from_memory). (2) If the head h has the EPP property and has a specifier or several, they are stored into the memory buffer (store_specs_into_memory). (3) If the head h misses a complement that it ought to have on the basis of its lexical features, the memory buffer is consulted and, if one is found, it will be merged to the complement position (fill_comp_from_memory). The phrase structure is explored, one head at a time, checking all three conditions for each head.

Option (1): fill in the SPEC position (fill_spec_from_memory). If h does not have specifiers, a constituent is selected from the memory buffer if and only if either (i) h has a matching specifier selection feature (e.g., v selects for DP-specifier) or (ii) h is an EPP head that requires the presence of phi-features in its SPEC that can be satisfied by merging a DP-constituent from the memory buffer (successive-cyclicity). Option (ii) is not yet fully implemented, as the generalized EPP mechanism involved (Brattico 2016; Brattico and Chesi 2019b) is not formalized explicitly. The condition, however, checks if phi-features are missing from the head but can be satisfied by merging a DP from the memory buffer. It is a "formal" specifier filling operation. An additional possibility is if the "specifier" of h is an adjunct: then an argument can be tucked in between the adjunct and the head, where it becomes a specifier, if conditions (i-ii) apply.

Option (2): store specifier(s) into memory. This operation is more complex because it is responsible for the generation of new heads if called for by the occurrence of extra specifiers. The operation takes place if and only if h is an EPP head: thematic constituents are not reconstructed by this operation. Let us examine the single specifier case first. A specifier is a complex left aunt constituent αP such that [αP [h(+EPP) XP]], αP has not been moved already somewhere else. If h has the generalized EPP feature PHI:0, αP will undergo A-reconstruction (local successive-cyclicity, Section 3.5.6); otherwise it will be put into memory buffer. The latter option leads possibly into long-distance reconstruction (A´-reconstruction). If αP has criterial features (which are scanned from it), formal copies of these features are stored to h. A formal copy of feature F is denoted by uF. If h is a finite head with feature FIN, a scope marker feature iF is created as well. This system means that F is the original criterial feature, uF is the formal trigger of movement, and iF is the semantically interpretable scope marker/operator feature. Lexical redundancy rules and parameters are applied to h to create a proper lexical item in the language being parsed (h might have language-specific properties). In addition, the label of h will be also copied to the new head, implementing the "inverse of feature inheritance." Finally, if h has a tail feature set, an adjunct will be created (create_adjunct). This is required when a relative pronoun creates a relative operator, transforming the resulting phrase into an adjunct.

If an extra specifier is found, the procedure is different. If the previous or current specifier is an adjunct and the correct specifier has no criterial features, then nothing is done: no intervening heads need to be projected. If there are two nonadjuncts, a head must be generated between the two, its properties copied from the criterial features of higher phrase and from the label of the head h (engineer_head_from_specifier)(41).The latter takes care of the requirement that when C is created from finite T, C will also has the label FIN. If the new head has a tail feature set, an adjunct is created (create_adjunct). This operation is required when a relative pronoun creates a relative operator, transforming the resulting phrase into a relative clause adjunct.

(41) $[_{\alpha P}$ F . . .] g^0 βP h^0



Option (3): fill in the complement position from memory (fill_comp_from_memory). A complement for head h is merged to Comp,hP from the memory buffer if and only if (i) h is a primitive head, (ii) h does not have a complement or h has a complement that does not match with its complement selection, and (iii) h has a complement selection feature that matches with the label of a constituent in the memory buffer.

Once the whole phrase structure has been explored, extraposition operation will be tried as a last resort if the resulting structure still does not pass LF-legibility (Section 3.5.7).

3.5.6 A-reconstruction

A-reconstruction is an operation in which a DP makes a local spec-to-spec movement, i.e. [DP_1 [α $\_\__1$ β]]. The operation is implemented if and only if α has the generalized EPP (PHI:0) property: we can assume that DP has been moved locally to satisfy this feature.

3.5.7 Extraposition as a last resort

Extraposition is a last resort operation that will be applied to a left branch α if and only if after reconstruction all movement α still does not pass LF-legibility. The operation checks if the structure α could be "saved" by assuming that its right-most/bottom constituent is an adjunct instead of complement. This possibility is based on ambiguity: a head and a phrase 'k + hp' in the input string could correspond to [K HP] or [K ⟨HP⟩]. The operation (try extraposition) will be tried if and only if (i) the whole phrase structure (that was reconstructed) does not pass LF-legibility test and (ii) the structure contains either finiteness feature or is a DP. Condition (i) is trivial, but (ii) restricts the operation into certain contexts and is nontrivial and possible must be revised when this operation is examined more closely. A fully general solution that applied this strategy to any left branch ran into problems. If both tests are passed, then the operation finds the bottom HP = [H XP] such that (i) HP is adjoinable in principle (is_adjoinable; (Brattico 2012))[2] and either (i.a) there is a head K such that [K HP] and K does not select HP or K obligatorily selects something else (thus, HP *should* be interpreted as an adjunct) or (i.b) there is a phrase KP such as [KP HP]. HP is targeted for possible extraposition operation. Next, it will be checked (redundantly?) that H is c-commanded by FIN or D and, if it is, HP will be promoted into adjunct (Section 3.5.8). This will transform [K HP] or [KP HP] into [K ⟨HP⟩] or [KP ⟨HP⟩], respectively. Only the most bottom constituent that satisfies these conditions will be promoted; if this does not work, and α is still broken, the model assumes that α cannot be fixed.

[2] The notion "is adjoinable" means that it can occur without being selected by a head. Thus, VP is not adjoinable because it must be selected by v.

To see what the operation does, consider the input string *John gave the book to Mary*. Merging the constituent one-by-one without extraposition could create the following phrase structure, simplifying for the sake of the example:

(42) John+ gave+ the book+ to Mary
[John [T [$_{DP}$[the book [P Mary]]]
SPEC P COMP

This interpretation is broken in several ways. First, it contains a preposition phrase [*the book* P DP], which is ungrammatical in English (by most analyses). Second, the verb *gave* now has the wrong complement PP when it required a DP. Extraposition will be tried as a last resort, in which it is assumed that the string 'D + N + [P + D + N]' should be interpreted as [DP ⟨HP⟩]. This will fix both problems: the verb now takes the DP as its complement (recall that the right adjunct is invisible for selection and sisterhood), and the preposition phrase does not have a DP-specifier. It is easy to see how the PP satisfies the criteria for the application of the extraposition operation: PPs are adjoinable, the configuration is [DP PP], and the PP is inside a FinP. The final configuration that passes LF-legibility test is (43).

(43) [John [gave [$_{DP}$[the book] ⟨to Mary⟩]]]

There is one more detail that requires comment: the labeling algorithm will label the constituent [DP ⟨PP⟩] as a DP, making it look like the PP adjunct were inside the DP. This is not the case: it is only attached to the DP geometrically, but is assumed to be inside the secondary syntactic working space. No selection rule "sees" it inside the DP. On the other hand, if this were deemed wrong, then the operation could attach the promoted adjunct into a higher position. This would still be consistent with the input '*the+book+to+Mary*'.

3.5.8 Adjunct promotion

Adjunct promotion (create adjunct) is an operation that changes the status of a phrase structure from non-adjunct into an adjunct, thus it transfers the constituent from the "primary working space" into the "secondary working space." The operation is part of reconstruction; decisions concerning what will be an adjunct and non-adjunct cannot be made in tandem with consuming words from the input. If the phrase targeted by the operation is complex, the operation is trivial: the whole phrase structure is moved. If the targeted item is a primitive head, then there is an extra concern: how much of the surrounding structure should come with the head? Ss it possible to promote the head to an adjunct while leaving its complement and specifier behind? It is obvious that the complement cannot be left behind, so that if H is targeted for promotion in a configuration [$_{HP}$ H, XP], then the whole HP will be promoted. This feature inheritance is part of the adjunct promotion operation itself. The question of whether the specifier must also be moved is less trivial. In the current implementation (which will possible require revision) it is assumed that if HP is in a canonical position as determined by its tail-head features and if it has the (!SPEC:*) feature, then whatever

constituent contains HP will be take with HP, thus it will contain the mandatory specifier. If there is no specifier, then the operation can still be done as the expression is crash either way. If HP is not in its canonical position, on the other hand, then the specifier is carried along if and only if (i) H does not have (–SPEC:*) feature explicitly banning a specifier and (ii) the label of the specifier L is not rejected by H (–SPEC:L). To see why this is relevant, consider the relative clause. A relative clause is headed by a relative operator at C. When the relative clause is promoted into an adjunct, we must promote the relative pronoun at Spec,CP as well, hence we promote the whole CP. The current implementation will not count specifiers, it only promotes the head, its complement and (given the above condition) the constituent that hosts HP.

3.6 Morphological and lexical processing

The parser-grammar reads an input that constitutes a one-dimensional linear string of elements at the PF-interface that are assumed to arise through some sensory mechanism (gesture, sound, vision). Each element is separated from the rest by a "word boundary." A word boundary is represented by space, although no space need to exist at the sensory level. Each such element at the PF-interface is then matched with items in the lexicon, which is a repository of a pairing between elements at the PF-interface and lexical items.

A lexical element can be *simple* or *complex*. A complex lexical element consists of several further elements separated by a #-boundary distinguishing them from each other inside phonological words. This correspond to a "morphologically complex word." A morphologically complex word cannot be merged to the phrase structure as such. It will be decomposed into its constituent parts, which will be matched again with the lexicon, until simple lexical elements are detected. A simple lexical element corresponds to a *primitive lexical item* that can be merged to the phrase structure and has features associated with it. For example, a tensed transitive finite verb such as *admires* will be decomposed into three parts, T/fin, v and V, each which is matched with a primitive lexical item and then merged to the phrase structure. Morphologically complex words and simple words exist in the same lexicon. A decomposition can be given also directly in the input. For example, applying prosodic stress to a word is equivalent to attaching it with a #foc feature. It is assumed that the PF-interface that receives and preprocesses the sensory input is able to recognize and interpret such features. The morphological parser will then extract the #foc feature at the PF-interface and feed it to the narrow syntax, where it becomes a feature of a grammatical head read next from the input.

It is interesting to observe that the ordering of morphemes within a word mirrors their ordering in the phrase structure. The morphological parser will therefore reverse the order of morphemes and features inside a phonological word before feeding them one by one to the parser-grammar. Thus, a word such as /admires/ → admire#v#T/fin will be fed to the parser-grammar as T/fin + v + admire(V). The same effect could be achieved by other means.

There are three distinct lexical components. One component is the language-specific lexicon (lexicon.txt) which provides the lexical items associated with any given language. Each word in this lexicon is associated with a feature which tells which language it belongs to. The second component hosts a list of universal redundancy rules (redundancy_rules.txt) which add features to lexical items on the basis of their category. In this way, we do not need to list in connection with each transitive verb that it must take a DP-complement; this information is added by the redundancy rules. The redundancy rules constitute in essence a 'mini grammar' which tell how labels and features are related to each other. The third component is a set of *universal lexical items* such as T, v, Case features, and many others (ug_morphemes.txt). When a lexical element is created during the parsing process, for example a C(wh), it must be processed through all these layers, while language must be assumed or guessed based on the surrounding context.

A lexical item is an element that is associated with a *set of features* that has also the property that it can be merged to the phrase structure. In addition to various selection features, they are associated with the label/lexical category (CAT:F), often several; phonological features (PF:F); a semantic *concept* interpretable at the LF-interface and beyond (LF:F)(of the type delineated by Jerry Fodor 1998); topological semantic field features (SEM:F); language features (LANG:F), tail-head features (TAIL:F, …G), probe-features (PROBE:F) and others. The number and type of lexical features is not restricted by the model.

3.7 EPP

A head h has the "EPP feature" if it has either [SPEC:*] or [!SPEC:*] or [+PHI] or [PHI:0]. The definition is disjunctive and unsatisfactory, which this reflects the fact that how to reduce the EPP remains an open question. Phi-features are involved in DP-movement and phi-agreement; the generalized specifier features represent the "generalized EPP".

3.8 LF-legibility

The purpose of the LF-legibility test is to check that the syntactic representation satisfies the LF-interface conditions and can be interpreted semantically. Only primitive heads will be checked. The LF-legibility test consists of several independent tests, as follows. Suppose we test head h. The *head integrity test checks* that h has a label. A head without label will be uninterpretable, hence it will not be accepted. A *probe-goal test* checks that a lexical probe-feature, if any, can be checked by a goal. Probe-goal dependencies are, in essence, nonlocal selection dependencies that are required for semantic interpretation (e.g. C/fin will select for T/fin over intervening Neg in Finnish). An *internal tail test* checks that D can check its case feature, if any. The *double specifier test* will check that the head is associated with no more than one (nonadjunct) specifier. The *semantic match test* will check that the head and its complement do not mismatch in semantic features. *Selection tests* will check that the lexical selection features of h are satisfied. This concerns all lexical selection features that state mandatory conditions (an adjunct can satisfy [!SPEC:L] feature). *Criterial*

feature legibility test checks that every DP that contains a relative pronoun also contains T/FIN. *Projection principle test* checks that argument (non-adjunct) DPs are not in non-thematic positions at LF. Discourse/pragmatic test provides a penalty for multiple specifiers (including adjuncts).

4 Using the program

4.1 Basic instructions

The code was written by using Python 3.x. and it consists of several modules, each in its own file. All module files must be in the same directory. You will also need the lexicon files mentioned in Section 3.6; they are part of the package. To use the parser, run the 'parse.py' script in that direction (by using some command prompt facility) by typing 'python.exe. parse.py'. You will have to configure the computer so that it will find the python interpreted (typically in a separate folder) when you type 'python.exe' (in Windows this will be specified as a path-variable). This script will read sentences from the file specified in the script itself (test_set_name), parses them, and produces two output files: file_name_results.txt, which contains the parsing output, and file_name_log.txt which contains the log. It will overwrite everything, so if you want to save the results, rename the files before running the parser again. The progress is also reported to the console.

4.2 Gold standards

The parser has one corpus titled 'gold standard corpus vX.txt' which contains a list of constructions and phenomena that the parser (version X) had been designed to process correctly. This file is matched with 'gold standard corpus vX_results_FINAL.txt' which contains the gold standard output. When you make modification to the parser, you would at some point run it with the gold standard corpus, produce a results file, and compare the results file with the gold standard corpus vX_FINAL.txt output. Comparisons should be done with an advanced editor instead of eyeballing; I use notepad++ that highlights all changes. This procedure will show you instantly how various theoretical ideas translate into concrete results (or non-results, as it often the case).

4.3 Version numbering

I use the program to do scientific (not practical) work. Each published work must be associated unambiguously with some version of the program, so that the simulations could be replicated. I use a system in which version 1.0 is associated with the first project/paper, 2.0 with the second, and so on. In this way, to replicate the results reported in the paper #1, it is assumed that version 1.0 was used. This version therefore exists as a fork. It does not matter if the paper is published or not, what matters if we want to produce a package that can be replicated in the future. The version that is used when work is being submitted is

typically something like x.9x, meaning that we are close to the final version x+1.0 for that project. All intermediate versions are numbered similarly.