Computational implementation of a linear phase parser. Framework and technical documentation (version 5.x)

2019

(Revised June 2020)

Pauli Brattico

Research Center for Neurocognition, Epistemology and Theoretical Syntax (NETS)
School of Advanced Studies IUSS Pavia

Abstract

This document describes a computational implementation of a minimalist linear phase parser-grammar. The parser-grammar assumes that the core computational operations of narrow syntax (e.g., Merge/Move/Agree) are applied incrementally and on a phase-by-phase basis 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).

How to cite this document: Brattico, P. (2019). Computational implementation of a linear phase parser. Framework and technical documentation (version 5.x). IUSS, Pavia.

Table of Contents

1 Introduction

This document describes the computational implementation of a linear phase parser that was originally developed and written by the author in an IUSS-funded research project between 2019-2020, in Pavia, Italy.[1] This document describes properties of the version 5.x, which keeps within the framework of the original work but provides a number of improvements, corrections and additions. My own philosophical interpretation of the model has also undergone slight development, as it was applied to an increasing number of empirical phenomena, and this is also reflected in this revised version. This document is written mainly for computer scientists or researchers who are interested in working with the concrete mathematical algorithm; the empirical substance is discussed in detail in published and unpublished literature. Thus, the exposition is simplified and structured as a tutorial that allows a competent programmer to implement the system by him- or herself. Many topics and problems that will interest linguists have been put aside to keep the exposition as simple as possible.

The document has three parts. Section 2 provides a brief tutorial to the software and its use. Section 3 provides a sketch the empirical framework that is presented in more detail in published papers. Section 4 goes into the details concerning full formalization and the computational implementation of that formalization. The quality of the material in this section varies from section to section depending strictly on the quality of the algorithm and empirical theory itself.

2 How to use the software

The linear phase hypothesis is a theory of language comprehension, implemented formally as a python (3.x) program. Its purpose is to test the theory rigorously. This testing consists in deriving by mechanical calculation whether the theory does capture all the data that is intended to capture. A typical use case might as follows. A researcher first selects an empirical topic of interest that he or she would like to model by using language comprehension as a framework. In the next step, the phenomenon is systematized into an explicit test corpus that contains the sentences, both grammatical and ungrammatical, that the theory must capture. A separate lexical file contains the words that appear in the test corpus, together with whatever lexical properties they are assumed to have. The main script of the program (*parse.py*) will then process the sentences in the test corpus, providing them with a grammaticality judgment (grammatical, ungrammatical, marginal), phrase structure analyses, semantic interpretation and step-by-step derivational log files. It will also produce phrase structure tree images of the solutions. Finally, it produces a lot of detailed information

[1] The research was conducted under the research project "ProGraM-PC: A Processing-friendly Grammatical Model for Parsing and Predicting Online Complexity" funded internally by IUSS (Pavia), PI Prof. Cristiano Chesi.

concerning the efficiency of the processing (garden paths etc.) The theorist will then examine the output and modify the theory accordingly, until a reasonably good match with experimental results is obtained.

The theory and the parser exist in the form of Python modules, which are text files containing the code. The models are all in the same directory. They use data from *language data working directory*, which should contain the test corpora and all other input files that are associated with separate studies. Thus, the *language data working directory* is further divided into several subdirectories, each associated with a particular study (published or not). The main script is started by writing *python.exe parse.py* into a command prompt facility that is in the directory containing the modules. The user must have Python (3.x) installed.

A word of caution is in order for those who are interested in the empirical subject matter. A scientific theory or hypothesis constitutes a guess of what might explain an empirical phenomenon. Within that framework, it makes little sense to copy a code that has been written as implementation of some particular theory or hypothesis authored by someone else and then use it in one's own work. That approach works in software engineering, but it is ill-suited for empirical science. A better approach is to examine all the evidence and available hypotheses and then write an implementation of one's *own* best guess, borrowing elements form here and there that are perceived as promising. This will bring an element of creativity into the process, and also allows the theorist to grasp of the empirical reasons any particular choice has been made.

3 The linear phase framework

3.1 Language comprehension and the linguistic architecture

A native speaker can interpret sentences in his or her language by various means, for example, by reading sentences printed on a paper. Since it is possible accomplish this task without external information, it must be the case that all information required to interpret a sentence in one's native language must be present in the sensory input. The operation that performs a mapping from linguistic sensory objects into sets of possible meanings is called the parser, or perhaps more broadly as *language comprehension*.

Some sensory inputs map into meanings that are hard or impossible to construct (e.g., *democracy sleeps furiously*), while others are judged by native speakers as ungrammatical. A realistic parser and a theory of language comprehension must appreciate these properties. The parser, when we abstract away from semantic interpretation, therefore defines a characteristic function from sensory objects into a binary (in some cases graded) classification of the input in terms of its grammaticality or some other notion, such as semanticality or marginality. These categorizations are studied by eliciting responses from native speakers. Any parser that captures this mapping correctly will be said to be *observationally adequate*, to follow the terminology from (Chomsky 1965). Many practical parsers are not observationally adequate: they do not distinguish ungrammatical inputs from grammatical inputs.

Some aspects of the parser are language-specific, others are universal and depend on the biological structure of the brain. A universal property can be elicited from the speakers of any language. For example, it is a universal property that an interrogative clause cannot be formed by moving an interrogative pronoun out of a relative clause (**who did John met the person that Mary admires_?*). On the other hand, it is a property of Finnish and not English that singular marked numerals other than 'one' assign the partitive case to the noun they select (*kolme sukkaa* 'three.sg.0 sock.sg.par'). The latter properties are acquired from the input during language acquisition. The universal properties plus the storage systems constitute the fixed portion of the parser, whereas language-specific contents stored into the memory systems during language acquisition and other relevant experiences constitute the language-specific, variable part. A theory of parser that captures the fixed and variable components in a correct or at least realistic way without contradicting anything known about the process of language acquisition is said to reach *explanatory adequacy*.

The distinction between observationally adequate and explanatorily adequate parser can be appreciated in the following way. It is possible to design an observationally adequate parser for Finnish such that it replicates the responses elicited from native speakers, at least over a significant range of expressions, yet the same parser and its principles would not work in connection with a language such as English, not even when provided a fragment of English lexicon. We could design a different parser, using different principles and rules, for English. To the extent that the two language-specific parsers differ from each other in a way that is inconsistent with what is known independently about language acquisition and linguistic universals, they would be observationally adequate but fall short of explanatory adequacy. An explanatory parser would be one that correctly captures the distinction between the fixed universal parts and the parts that can change through learning, given the evidence of such processes, hence such a parser would comprehend sentences in any language when supplied with the (1) fixed, universal components and (2) the information acquired from experience. We are interested in a theory of language comprehension that is explanatory.

Suppose we have constructed a theory of the parser that is, or can be argued to be, observationally adequate and explanatory. Then it is possible to ask a further question: does it agree with the data obtained from neuro- and psycholinguistic experimentation? Realistic parsing involves several features that an observationally adequate explanatory theory of the parser need not capture. One such property concerns automatization. Systematic use of language in everyday communication allows the brain to automatize the recognition and processing of linguistic stimuli in a way that an observationally adequate explanatory parser might or might not want to be concerned with. Furthermore, real language processing is sensitive to top-down and contextual effects that can be ignored when constructing a theory of the parser. That being said, the amount of computational resources consumed by the parser should be related in some meaningful proportion with what is observed in reality. If, for example, the parser engages in astronomical garden-path derivations when no native speaker exhibits such problems, then the parser can be said to be insufficient in its ability to mimic real language comprehension. Let us say that if the parser's computational efficiency

matches with that of real speakers, the parser is also *psycholinguistically adequate*. I will adopt this criterion in this study as well.

Language production utilizes motoric programs that allow the speaker to orchestrate a complex motoric sequence for the purposes of generating concrete speech or other forms of linguistic behavior; language comprehension involves perception and not necessarily concrete motoric sequencing. Although there is evidence that perception involves (or "activates") the motoric circuits and vice versa, overt repetition is (thankfully) not a requirement. A more interesting claim would be that the two systems share computational resources. This is the position taken in this study. Specifically, I will hypothesize, following (Phillips 1996, 2003) but interpreting that work in a slightly weaker form, that many of the core computational operations involved in language production and/or in the determination of linguistic competence are also involved in language comprehension. The same could be true of lower-level language processing, so that the motoric generation of, say, phonemes is decoupled in the human brain with systems that are ultimately responsible for the perception of the same units. The idea is illustrated in Figure 10.

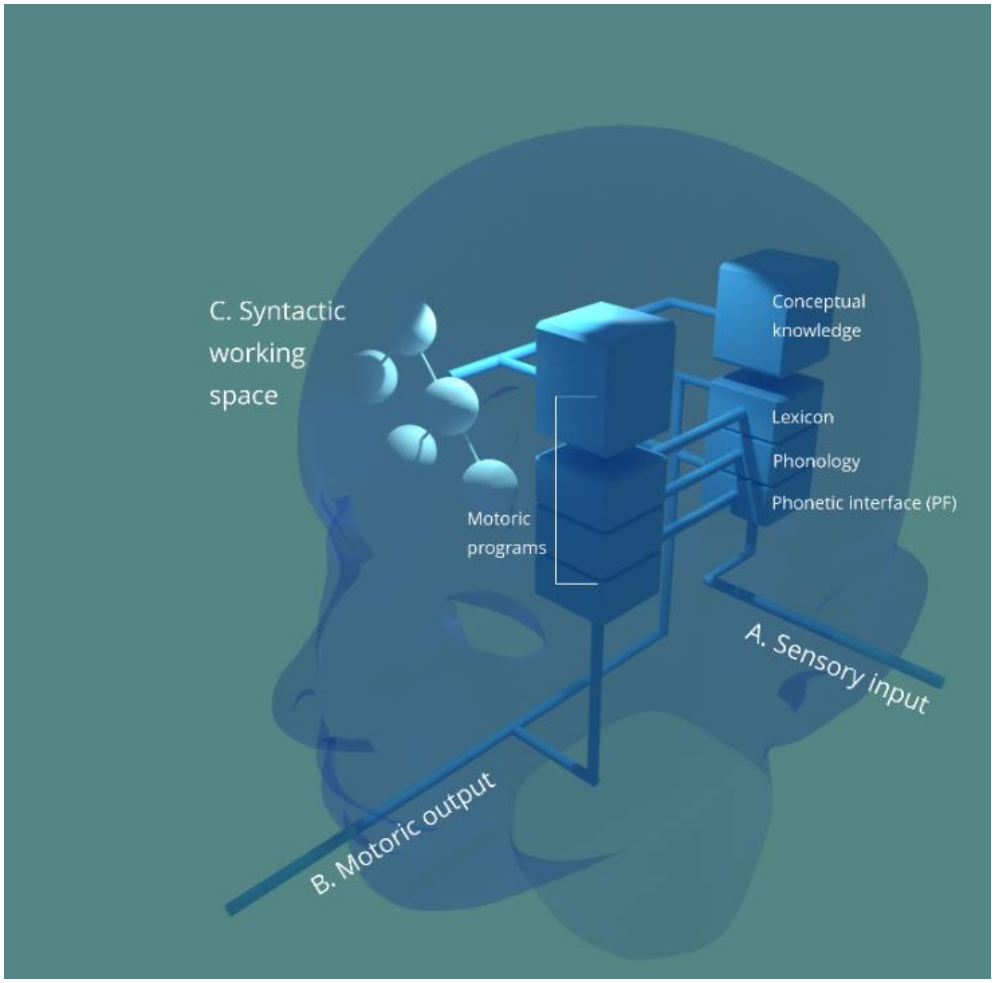


Figure 10. Language comprehension and production are based on the same system. The system is based on sensory mechanisms (posterior parts) and motoric mechanism (anterior parts) which work in tandem in both

language production and comprehension, hence the connections between the systems. Both connect with a syntactic working space utilizing core computational information processing operations (Merge, Move, Agree). The syntactic working space seems to be realized by the neuronal circuits of the frontal cortex and its reciprocal connections to the subcortical regions. This architecture, although simplified, seems to me to be at least consistent with what we know today based on neurolinguistic research.

Language comprehension proceeds from concrete sensory properties into abstract meaning. This follows from the assumption that the process begins from sensory input and can operate successfully without additional input. Research shows that the process relies on increasingly more abstract properties of the input. Thus, once the sensory input is judged as linguistic (and not, say, music or ambient noise), it will be interpreted in terms of phonemes, syllables, surface morphemes and features, phonological words, lexical items, syntactic phrase fragments and whole expressions, and finally in terms of meaning and communicative intentions. More abstract interpretations and categories rely on those made at earlier stages.

3.2 Merge-1

Linguistic input is received by the hearer in the form of sensory stimulus. We think of the input as a one-dimensional string $\alpha * \beta * \ldots * \gamma$ of phonological words. 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 hierarchical relations between words. 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 (Phillips 1996, 2003). For example, if the input consists of two words $\alpha * \beta$, Merge yields $[\alpha, \beta]$(1).

(1) John * sleeps.
↓ ↓
[John, sleeps]

The assumption that this process is incremental or "linear" means that each word consumed from the input will be merged to the phrase structure as it is being consumed. No word is 'put aside' for later processing. 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*]]

The operation illustrated in (2) differs in a number of respects from what constitutes the standard theory of Merge at the present writing. I will label the operation in (2) by Merge-1, the symbol -1 referring to the fact that we look the operation from an 'inverse perspective': instead of generating linear sequences of words by

applying Merge, we apply Merge-1 on the basis of a linear sequence of words and thus deriving the structure backwards.

Several factors regulate Merge-1. 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 left-to-right depth-first linearization algorithm would produce **John furiously sleeps* from (2)a. Therefore, this alternative can be rejected on the grounds that the result is ungrammatical and is not consistent with the word order discovered from the input.

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.

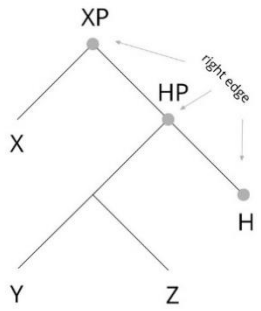


Figure 2. The right edge of a phrase structure. These three nodes are all possible attachment sites for an incoming word. X, Y and Z are not. This condition follows from the assumption that linearization in language production is always top-down/left-right.

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. It must be possible to backtrack and re-evaluate a parsing decision made at an earlier stage. Let us assume that all legitimate merge sites are ordered and that these orderings generate a recursive search space that the parser-grammar use 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. All solutions constitute 'potential phrase structures', which are ordered in terms of their ranking, while one of them is selected. This is illustrated in Figure 2.

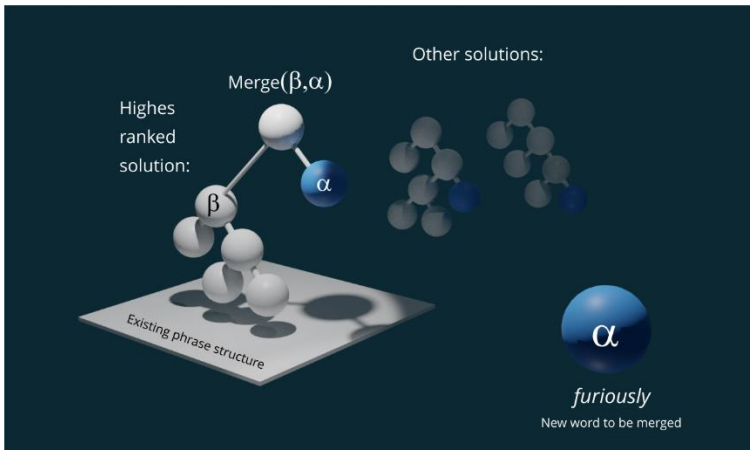


Figure 2. All possible merge sites of α on the right edge of the existing phrase structure β are ordered and then explored in that order. This operation takes place in the syntactic working space (see Figure 10).

Both solutions (a) and (c) are "countercyclic": they extent the phrase structure at its right edge, not at the highest node. A countercyclic Merge-1 is more complex than simple merge that combines to constituents: it must insert the constituent into a phrase structure and update the constituency relations accordingly (Section 4.6.1). This type of derivation could be 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 to the highest node. It is more correct to say that merge is "to the right edge." In literal top-down grammars, such as that of (Chesi 2012), no bottom-up operation, to the right edge or to the left edge, is allowed.[2]

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).

[2] Except for the root. The key empirical idea in restricted/literal top-down grammars is that every merge operation must be selected or expected "from above." That condition is too strong for parsing, which operates with a PF-object, but it might be formulated as a condition for the LF-interface. The present model takes a weaker but also more general position, according to which top-down merge is possible but not mandatory.

(4) John sleeps furiously

↓ ↓ ↓

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

During the first stage, 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 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 locally 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 3.4. It is important to keep in mind that in the Phillips architecture constituency relations established at point *t* do not necessarily hold at a later point *t* + *n*.

There are at least two ways to think about what these changes mean to the overall linguistic architecture. 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 give up standard properties of Merge that are postulated based on 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. The weaker hypothesis is less interesting than the strong one, and less parsimonious as well, but it makes it possible to pursue the comprehension 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 can "converge" into some core set of assumptions or, at the very least, that they are not mutual contradictory. Development of such a unified theory is the goal that the present work tried to contribute. The operation of left-to-right Merge and its role in the whole language architecture is illustrated in Figure 11.

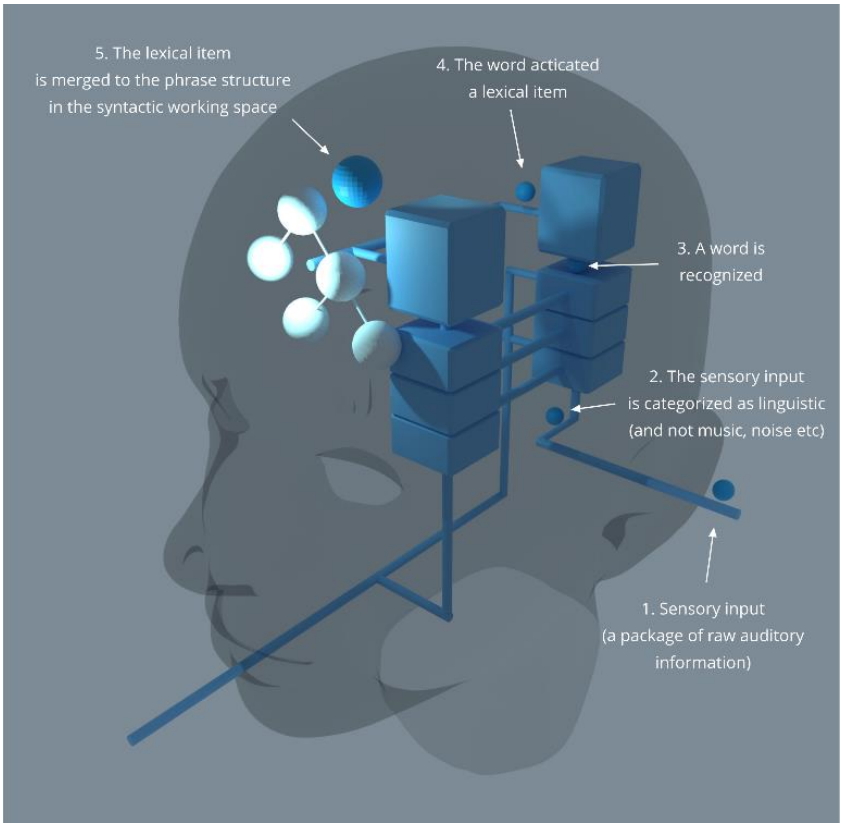


Figure 11. Merge and its role in language comprehension. Incoming sensory input is first analyzed through several subsystems until a phonological word is recognized. The phonological word is then decomposed into its primitive parts (if any), which are matched with lexical items in the lexicon. A lexical item (essentially a set of features) is then merged to the existing phrase structure in the syntactic working space.

It is not necessary to empower the parser with filtering and ranking. If they are not included, then the parser will explore all possible combinations. The result is a parser that can be observationally adequate and even explanatory, but psycholinguistically vastly implausible. A parser that has no filtering or ranking function of any kind will typically use astronomical amount of computational resources when parsing a simple sentence. Therefore, filtering and ranking should be included if only for practical reasons. Once they have been included, it is then possible to test alternative ranking methods and match evaluate them against data from psycholinguistic experiments.

3.3 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) John admires Mary
↓ ↓ ↓
[John [admires Mary]]

There is linguistic evidence that this derivation matches with the correct hierarchical relations between the three words (ignoring, of course, many details). 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 admiring, John will be the agent. If we change the positions of the arguments, the interpretation reverses. 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 previously by (Chesi 2004, 2012) within the framework of a top-down grammar, the key idea being that lexical features are ways to form and satisfy top-down "expectations." The fact that *admire* has the lexical feature !COMP:D can be used by Merge^{-1} to create a ranking based on an expectation: when *Mary* is consumed, the operation checks if any given merge site allows/expects 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	*Mary* satisfies an "expectation"

Feature COMP:L means that the lexical item *licenses* 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 trying to sort out an input, it uses the lexical features (among other factors) to rank solutions. These features cannot always be used to filter out possible solutions, because constituency relations can change in the derivation's future (Section 3.2). But when the phrase structure has been completed, and there is no longer any input to be consumed, the same features can be used for filtering purposes. At this point we know that no further rearrangement will take place. A functional head that requires a certain type of complement (say v-V) will crash the derivation if the required complement is missing. This procedure concerns features that are positive and mandatory (e.g. !COMP:V or negative –COMP:V). This filtering operation is performed at the LF-interface (discussed later) and will be later called an "LF-legibility test." Its function is to make sure that the phrase structure can be interpreted by the conceptual-intentional systems. But little filtering can be done locally while the first pass parse structure is being derived from the surface string.

Let us return to the example with *furiously*. What might be the lexical features that are associated with this item? The issue depends 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 (i.e. *Mary* has a lexical selection feature –COMP:ADV). We are left with two options (8)a-b.

(8)

a. [[$_{S}$ John [admires Mary]] furiously]

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

Independently of which one of these two solutions is the more plausible one (or if they both are equally plausible), we can guide Merge^{-1} by providing the adverbial with a lexical selection feature which determines what type of specifiers/sisters it is allowed or is required to have. I call such features *specifier selection features*. A feature SPEC:S ("select the whole clause as a specifier/sister") favors solution (a), SPEC:V favors solution (b). If the adverbial has both features, or has neither, then selection is free.

Notice that what constitutes a specifier selection feature will guide the selection of a possible left sister during parsing. Because constituency relations may change later, we do not know which elements will form specifier-head relations in the final output. Therefore, we use specifier selection to guide the parser in selecting left sisters, and later use them to verify that the output contains proper specifier-head relations. To illustrate, consider the derivation in (9).

(9) John's brother admires…

↓ ↓

[John's brother] T(v, V) = finite tensed transitive verb

The specifier selection feature of the finite transitive verb will instruct the parser to merge the finite transitive verb to the right of the subject *John's brother*. Notice that at the final output, after the processing of the direct object, the two are no longer sisters; instead, we will have the following configuration:

(10) John's brother admires Mary

↓ ↓ ↓

[[John's brother] [T(v,V) DP]]

The grammatical subject occurs in the canonical specifier position of T(v,V). It is important to keep in mind, once again, that in this framework most configurations established during first pass parsing are subject the change later in the parsing derivation.

3.4 Phases and left branches

Let us consider next the derivation of a slightly more complex clause (11).

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

After the finite verb has been merged with the DP *John's mother*, no future operation can affect the internal structure of that DP. 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 by (Brattico and Chesi 2020). We can formulate the condition tentatively as (12), but a more rigorous formulation will be given as we proceed.

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

All left branches are "thrown away" from the working space once have been assembled and fully processed (13).

(13) John's + mother + admires + Mary
[John's mother]
↓ + admires + Mary
(has been sent out)

If no future operation is able to affect a left branch, 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. 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. Any merge site that leads into an ungrammatical or uninterpretable left branch can be either filtered out as either unusable or be ranked lower.

The notion that each left branch phase is processed independently requires a further comment. A parser implements a function from PF-objects into sets of LF-objects, semantically interpretable phrase structure objects. Therefore, each left branch must be transferred from the active syntactic working space to the LF-interface, and from the LF-interface, if the representation is well-formed, into the conceptual-intentional system in which it is provided a semantic interpretation. The process that maps a candidate left branch phase into the LF-interface is called *transfer*. It involves several mechanical, reflex-like operations that "normalize" the phrase structure for semantic interpretation, for example, by reconstructing operator movement, so that each argument may receive a thematic role in addition to marking the scope an operator. These processes are discussed later, but the general architecture is illustrated in Figure 5.

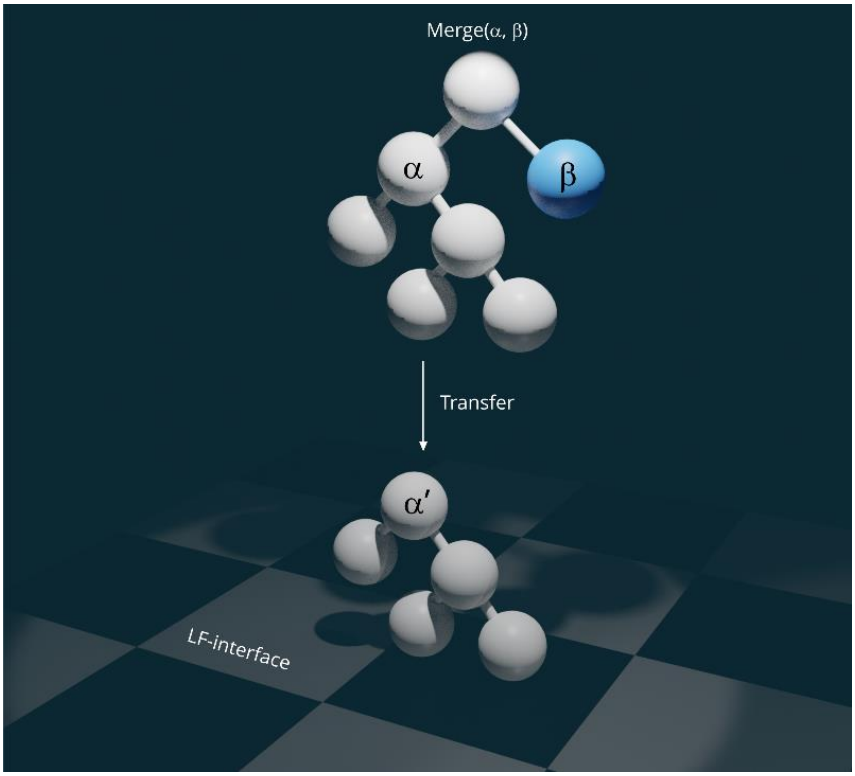


Figure 5. Transfer transforms a candidate left branch phrase α to the LF-interface. The transfer operation implements several computational operations, and the resulting phrase structure α´ is then evaluated at the LF-interface for semantic legibility. If the phrase structure is legit, it will be passed on to the conceptual-intentional systems for semantic interpretation. Transfer will be modeled here as a normalization or an error correction/tolerance algorithm,

The nature of the LF-interface is not trivial. It is typically conceptualized as the "interface" between syntax and semantics. One condition is that the syntactic representation occurring at the LF-interface must be interpretable by the conceptual-intentional (semantic systems) in the way a native speaker interprets the sentence. Once the representation passes at LF, it will be handed over to extrasyntactic systems and it disappears from syntax.

3.5 Labeling

Suppose we reverse the arguments (11) and derive (14).

(14) Mary admires John's mother

↓ ↓ ↓ ↓

Mary [admires [John's mother]]

COMP:D

In this configuration the verb selects for a DP, but the complement selection feature refers to the label D of the complement. What is relationship between the label D and the phrase that occurs in the complement position of the verb in the above example? That relationship is defined by a recursive labeling algorithm (15). The algorithm searches for the closest possible primitive head from the phrase, which will then constitute the label. Here "closest" means closest form the point of view of the selector; if we look at the situation from the point of view of the labeled phrase itself, then closest is the "highest" or "most dominant" head.

(15) Labeling

Suppose α is a complex phrase. Then

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

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

c. if the right constituent is not an adjunct, apply (15) recursively to it; otherwise,

d. apply (15) 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*. A constituent that has only the left or right constituent, but not both, will also be primitive according to this definition. Conditions (15)c-d mean that labeling – and hence selection – ignores right adjuncts (this will be a defining feature of "adjunct"[3]).

Consider again the derivation of (1), repeated here as (16).

(16) John + sleeps.

↓ ↓

[John, sleeps]

If *John* is a primitive constituent having no left or right daughters, labeling will categorize [*John sleeps*] as a DP. The primitive left constituent D will be its label. This is wrong: (16) is a sentence or verb phrase, not a DP. One solution is to reconsider the labeling algorithm. That seems implausible: (15) captures what looks to be a general property of language, thus this alternative would require us to treat (16) and other similar examples as exceptions. Yet, there is nothing exceptional or anomalous in (16). The representation should come out as a VP, with the proper name constituting an argument of the VP. Thus, the proper name should not constitute the (primitive) head of the phrase. I assume that *John* is a complex constituent despite of appearing as if it were not. 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

[3] 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.

decomposed into D + N structure. The structure of (16) is therefore (17). The lexical-morphological component will be discussed later.

(17)
John + sleeps.
↓ ↓
D N sleeps
↓ ↓ ↓
[$_{VP}$[$_{DP}$D N] sleeps]

It is important to note that labeling 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 (17), which means that the whole phrase will come out as a verb phrase. The outcome will be the same if the verb is transitive. The structure and label are provided in (18).

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

This analysis presupposes that there is some way to unpack *John* into the complex constituent [D N]. That operation is part of transfer (Section 3.8.3).

3.6 Adjunct attachment

Adjuncts, such as adverbials and adjunct PPs, present a separate problem. Consider (19).

(19)

a. *Ilmeisesti* Pekka ihailee Merjaa.
apparently Pekka admires Merja

b. Pekka *ilmeisesti* ihailee Merjaa.
Pekka apparently admires Merja

c. Pekka ihailee *ilmeisesti* Merjaa.
Pekka admires apparently Merja

d. ??Pekka ihailee Mejaa *ilmeisesti*
Pekka admires Merja apparently

The adverbial *ilmeisesti* ‘apparently’ can occur almost in any position in the clause. Consequently, it will be merged into different positions in each sentence. This confuses labeling. The problem is to define what this type of ‘free attachment’ means. It is assumed here that adjuncts are geometrical constituents of the phrase structure, but they are stored in a parallel syntactic working memory and are invisible for sisterhood, labeling and selection in the primary working memory. Thus, the labeling algorithm specified in Section 3.5 ignores adjuncts. The label of (20) becomes V: while analyzing the higher VP shell, the search algorithm does not enter inside the adverb phrase; the lower VP is penetrated instead (15)c-d.

(20) H^0 [$_{VP}$ John [$_{VP}$[$_{VP}$ admires Mary] ⟨$_{AdvP}$ furiously⟩]]

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 (20). Consider (21).

(21) John [sleeps ⟨$_{AdvP}$ furiously⟩]

The adverb 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 parallel syntactic working memory and is only 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 (21) 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). Adjunct attachment is illustrated in Figure 3.

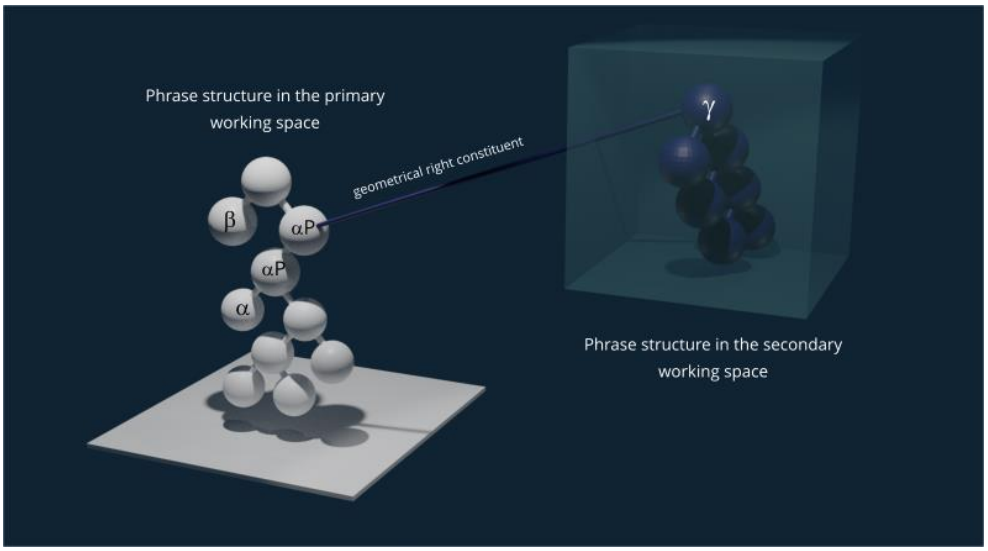


Figure 3. Adjuncts γ are attached to the phrase structure as geometrical constituents, but they are "pulled" into a secondary syntactic working space. Being in the second working space, they are invisible for the primary computational operations (sisterhood, selection, labeling) inside the primary working space. Element β therefore sees αP as its sister. However, they are still part of the phrase structure and can be targeted for case assignment and movement.

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. It follows that adjuncts can be merged anywhere, which is not correct. I assume that each adverbial (head) is associated with a feature *linking* it with a feature in its hosting, primary structure. The linking relation is established by an 'inverse probe-goal relation' 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. 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 (22). This theory owes much to (Ernst 2001).

(22) 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.

Condition (i) is uncontroversial. Condition (ii) allows adverbials to remain in a low right-adjoined or extraposed positions in a canonical structure. If condition (ii) is removed, adverbials will be reconstructed into positions in which they are inside the corresponding projections (reconstruction will be discussed later). A VP-adverbial will be reconstructed inside a VP, and so on. Some versions of the theory keep (ii), others deny it; the matter is controversial. The tail-head dependency is illustrated in Figure 4.

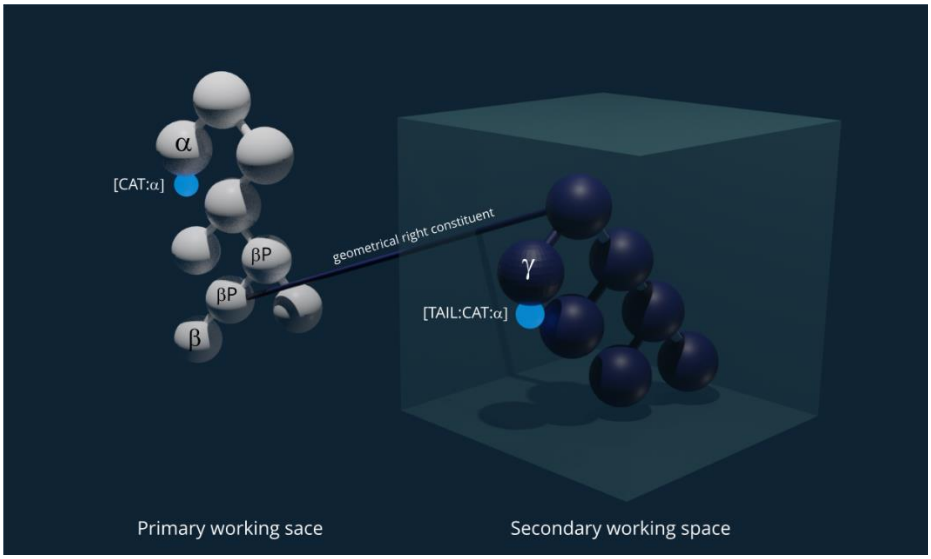


Figure 4. Adjuncts must satisfy a tail-head relation. The head γ of the adjunct with the tail-head feature [TAIL:CAT:α] must be c-commanded by a head that has feature [CAT:α]. In the more traditional bottom up theories, [TAIL:CAT:α] is the goal feature requiring checking, whereas [CAT:α] is the probe feature. Another legitimate condition is if the probe feature were at β, because the adjunct would then be inside βP.

If an adverbial/head does not satisfy a tail feature, it will be reconstructed into a position in which it does during transfer. This operation will be discussed in Section 3.8.4.

Adjuncts can be visualized as constituents that reside in a parallel syntactic working space, being attached to the main structure more loosely. The idea that they reside in a parallel working space is supported by the fact that the computational operations (labeling, sisterhood) targeting the primary hosting structure do not see them, and by the fact that the mechanism is iterative in the sense that an adjunct can have its own further

adjuncts. Adjuncts are also transferred to LF independently, as independent phases. Their connection to the hosting primary structure is loose in the sense that they are only linked with a feature in the primary structure. We can think of adjuncts as increasing the "dimension" of the phrase structure, in the sense that any head or feature in the primary structure can be shadowed by an adjunct that is silently linked with it. Finally, the linking relations are typically or always interpreted as predication: the adjunct phrase expresses a predicate, which is attributed to the head/feature with which it is linked with. A VP-adverbial, for example, attributes a property to the event denoted by the V.

3.7 EPP and "unselective" selection

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:*, SPEC:* and !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 3.8.1). This constitutes a sufficient (but not necessary) feature for reconstruction; see 3.8.1. Language uses phrasal movement, and hence an unselective specifier feature, to represent a null head (such as C) at the PF-interface. Corresponding to SPEC:* we also have !COMP:*, which is a property all functional heads have, possibly by definition.

3.8 Move-1

3.8.1 A-bar 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. For example, regular referential or quantificational arguments must occur inside the verb phrase at the LF-interface in order to receive thematic roles and satisfy selection. Example (23) is a sentence in which all elements occur in their canonical positions in the input: the parser reaches a plausible output by merging the elements into the right edge as they are being consumed.

(23) John admires Mary
↓ ↓ ↓
[John [admires Mary]]

Example (24) shows a variation in which this is no longer the case.

(24) Ketä Pekka ihaile-e __? (Finnish)
who.par Pekka.nom admire-3sg
'Who does Pekka admire?'

The parser will generate the following first pass parse for this sentence (in pseudo-English for simplicity):

(25) [who [Pekka admires]]

The clause violates selection: there are two specifiers DP_1 and DP_2 at the left edge, and *admire* lacks a complement. The two violations are related: the element that triggers the double specifier violation at the left edge is the *same element* that is missing from the complement position. We therefore know that the interrogative pronoun causes a double specifier error because it has been "dislocated" to the noncanonical, "wrong" position from its canonical complement position, and this is the reason it also triggers complement selection failure. The parser must therefore reverse-engineer dislocation in order to create a representation that can be interpreted at the LF-interface. These operations, which is called *reconstruction*, take place during transfer.

Before addressing the specific formal mechanisms, it makes sense to ask a more fundamental question: why do many languages dislocate words and phrases? We can approach this question from the following angle. The first-pass parse generated directly from the input string by merging constituents to the right will be related systematically and transparently to the linguistic sensory object itself, as the two are mapped to each other by a simple algorithm. Specifically, we can always generate the final sensory object by applying a left-to-right depth-first linearization algorithm to the first-pass parse. We could emphasize this aspect by calling the first-pass parse also as the *spellout structure*. The spellout structure therefore captures an abstract sensorimotoric plan that captures the way language "packages" linguistic information for the purpose of linguistic communication. Obviously, however, languages also differ from each other in how they do this packaging: Finnish does it differently than English. The conceptual systems, however, are often assumed to be near-universal. It is possible to translate English sentences into Finnish, and vice versa, which suggests that the underlying conceptual representations are closely related, making communication between speakers of different languages possible and effortless, at least after linguistic packaging has been acquired and automatized. This requires that the abstract sensorimotoric plans are converted into a format from which most language-specific properties have been eliminated. This normalization operation is performed by transfer. If this is true, then dislocation – in which there is considerable variation between languages– has to do with sensorimotoric packaging of linguistic information. I return to this issue later after considering some of the details of the implementation itself.

In Finnish, a standard interrogative creates a spellout structure in which the sentence begins with two DPs. This double specifier problem is first "fixed" by generating a head between them (26), the interpretation being that the *reason* the interrogative phrase was moved in the first place was that the head C(wh) itself is phonologically null, thus the phrase is moved to focus the listener's attention to the interrogative feature defining the scope of the question.

(26) [Ketä [C(*wh*) [Pekka ihailee __]]]?
wh C(*wh*)
who.par Pekka admires
'Who does Pekka admire?'

The mechanism involves two steps. First we must recognize that a head is missing. That is inferred from the existence of the two specifiers of the T/fin head (two specifiers are not allowed unless the head as a specified feature licensing them). The next step is to generate the label/feature +*wh* for the head. This information is obtained from the *criterial feature*, i.e. from the fact that the element is an interrogative pronoun. This allows the parser-grammar to infer both the *existence* and the *nature* of the phonologically null head and thus infer that this is an interrogative clause with the scope marked by C(*wh*). The interrogative phrase must then be reconstructed back to its canonical LF-position to satisfy the complement selection for the verb *admire*.

(27) Who does John admire __?
——— Reconstruction ———→

Reconstruction (called Move-1) works by copying the element occurring in the wrong position and by "dropping" or reconstructing it into the same structure, in this case reconstruction begins from the sister of the targeted element and proceeds downward while ignoring left branches (phases) and right adjuncts. The element is copied to the first position in which (1) it can be selected, (2) is not occupied by another legit element, and in which (3) it does not violate any other condition for LF-objects. If no such position is found, the element remains in the original position and may be targeted by another operation during transfer. If the position is found, it will be copied there; the original element will be tagged so that it will not be targeted for the second time.

Move-1 takes place during transfer. Transfer itself is triggered under three conditions. One condition is that all words have been consumed, in which case the final product will be transferred to LF. The second condition occurs upon Merge-1 (α, β) and leads to transfer of α. The third condition is that α is interpreted asn an adjunct. The two first conditions are expressed by (28)(Brattico and Chesi 2020), while the third was added later.

(28) Transfer(α) is and only if either
i. Merge(α, β) or
ii. α is completed or
iii. α is an adjunct.

The reason α must be transferred due to (i) is that we want to check if it constitutes a legitimate LF-object. If not, the fragment/sentence can be rejected. Thus, if we are considering Merge-1 (α, β) as a possible merge

site for β, failed transfer can be interpreted a signaling that the solution should be filtered out or ranked lower. If α is complete and transfer fails, then the parser must backtrack.

One question that remains without explicit answer is the explicit trigger for the operation that reconstructs the interrogative pronoun to the complement position. Notice that after C(*wh*) has been generated to the structure, all selection features are potentially checked. One possibility is that the operation is triggered by the missing complement. In the present implementation it is assumed that error recovery is triggered at the site of the element that needs reconstruction. In this case, it is assumed that C(*wh*) has an unselective specifier selection feature SPEC:* (=generalized EPP feature, or second edge feature in the sense of (Chomsky 2008)) which says that the element may have a specifier with any label that is not selected (in the present formalization, labels select and are selected). Reconstruction is triggered by the presence of this feature (Brattico and Chesi 2020). The reconstructed canonical position will be found during reconstruction as it takes place during transfer.

3.8.2 A-reconstruction and EPP

One sufficient condition for phrasal reconstruction is the occurrence of a phrase at the specifier position of a head that has the SPEC:* feature (=EPP in the standard theory). This alone will trigger reconstruction, with or without criterial features. If there are no criterial features, then A-movement reconstruction is activated which moves the element into the specifier position of the next projection downstream.

3.8.3 Head reconstruction

Many heads occur in noncanonical positions in the input string. Consider (29).

(29) Nukku-a-ko Pekka ajatteli että hänen pitää _?
sleep-T/inf-Q Pekka thought that he must
'Was it sleeping that Pekka thought that he must do?'

The complex word *nukkua-ko* 'sleep-T/inf-Q' contains elements that are in the wrong place. The infinitival verb form (T/inf, V) cannot occur at the beginning of a finite clause, and there is an empty position at the end of the clause in which a similar element is missing.

Lexical and morphological parser 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 in the order illustrated by (30). Symbol "#" indicates that there is no word boundary between the morphemes/features.

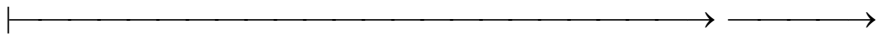
(30) C(-kO) + #T_{inf} + #V + Pekka + ajatteli + että + hänen + pitää
C T_{inf} V Pekka thought that he must

The individual heads are then collected into one complex head by the syntactic parser. Specifically, the incoming morphemes are stored into the right constituent of the preceding morpheme which creates a linearly ordered sequence of primitive morphemes.[4] Thus, if syntax receives a word-internal morpheme β, it will be merged to the right edge of the previous morpheme α: $[_{\alpha}\ \varnothing\ \beta]$. Notice that by defining "complex constituent" as one that has both the left and right constituent, $[_{\alpha}\ \varnothing\ \beta]$ comes out as primitive and constitutes a head of a projection. The linear sequence C-T/inf-V becomes $[_{C}\ \varnothing\ [_{T/inf}\ \varnothing\ V]]$. This is what gets first-merged^{-1} (31).

(31) $[_{C}\ \varnothing\ [_{T/inf}\ \varnothing\ V]]$ Pekka ajatteli että hänen pitää __.
Pekka thought that he must

Representation (31) is not a legit LF-object. The first constituent $[_{C}\ \varnothing\ [_{T/fin}\ \varnothing\ V]]$ cannot be interpreted, and the embedded modal verb *pitää* 'must' lacks a complement. Both problems are solved by *head reconstruction* which drops $[_{T/inf}\ \varnothing\ V]$ from within C into the structure. This is done by finding the closest position in which T/inf can be selected and in which it does not violate local selection rules. The closest possible position for T/inf is the empty position inside the embedded clause. V will then be extracted in the same way and reconstructed into the complement position of T_{inf}.

(32) $[_{C}\ \varnothing\ [_{T/inf}\ \varnothing\ V]]$ Pekka ajatteli että hänen [pitää $[[_{T/inf}\ \varnothing\ V]$ V]].
Pekka thoughr that he.gen must



The operation fixes problems with cluttered heads in the input. It resembles phrasal movement reconstruction: it considers the cluttered components of the complex head to be in a wrong position and attempts to drop them into first positions available in which they could create a legitimate LF-object.

3.8.4 Adjunct reconstruction

Consider the pair of expressions in (33) and their canonical derivations.

(33)

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

[4] Pronominal clitics are right/left constituents that are complex but occur without a sister.

[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 (b) is incorrect. Native speakers interpreted the thematic roles identically in both examples. The subject and object are again in wrong positions. Yet, 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/leftward in order to reach the canonical LF-position Spec,VP.

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, 2019c, 2019b). See (Baker 1996; Chomsky 1995: 4.7.3; Jelinek 1984) for similar hypothesis. This can be captured in the following way. Suppose that case features must establish local tail-head (inverse probe-goal) relations with functional heads as provided, tentatively and for illustrative purposes only, in (34).

(34) Case features must establish local tail-head relations such that
a. [NOM] is checked by +FIN;
b. [ACC] is checked by +ASP/v;
c. [GEN] is checked by −FIN;
d. [PAR] is checked by −VAL.

Case forms are, therefore, morphological reflexes of tail-head features. The symbol "−VAL" refers to a head that never exhibits φ-agreement, but what the features are is not crucial here; what matters is that case features are associated with c-commanding functional heads and features therein. 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 (34) is satisfied. In this way, the inversed subject and object can find their ways to the canonical LF-positions (35). Notice that because the grammatical subject *Pekka* is promoted into adjunct, it no longer constitutes the complement; the partitive-marked direct object does.

(35) [⟨Merjaa⟩$_2$ T/fin [__$_1$ [käski [meidän [ihailla [__$_2$ ⟨Pekka⟩$_1$]]]]
+FIN ⟵NOM −VAL ⟵PAR
Merja.par asked we.gen to.admire Pekka
'Pekka asked us to admire Merja.'

The operation is licensed by rich case morphosyntax, which allows transfer to dislocate the orphan arguments to their correct canonical positions. In a language with richer morphosyntax, many more errors in word order are possible than in a language in which morphosyntax is more meager. Case features are an extension of the tail-head dependency feature introduced earlier in Section 3.6.

3.8.5 Ordering of operations

Movement reconstruction is part of transfer. The operations must be ordered. Both A´/A-reconstruction and adjunct reconstruction presuppose head reconstruction, as 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. The whole sequence is (36).

(36) Merge-1(α, β) $\rightarrow$ Transfer α (reconstruct heads $\rightarrow$ reconstruct adjuncts $\rightarrow$ reconstruct A/A'-movement)

The sequence is performed in a one fell sweep, in a reflex-like manner; it is not possible to evaluate the operation only partially or backtrack some moves while executing others.

3.9 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 surface vocabulary (dictionary) first matches phonological words with morphological decompositions. The decomposition consists of a linear string of morphemes m_1#...#m_n that are separated and inserted into the linear input stream individually (37). Notice the reversed order.

(37)	Nukku-u-ko	+	Pekka	$\rightarrow$	Q	+	T/fin	+	V	+	Pekka
	sleep-T/fin-Q		Pekka		$\downarrow$		$\downarrow$		$\downarrow$		$\downarrow$
	sleep#T/fin#Q				Merge^{-1}		Merge^{-1}		Merge^{-1}		Merge^{-1}
	'Does Pekka sleep?'										

The lexical entry for the complex phonological word *nukkuako* is therefore 'sleep#T/fin#Q', where each morpheme *sleep-*, T/fin and Q are matched with lexical items (LIs) in the same lexicon. The lexicon has a hierarchical structure: one lexical entry can contain pointers to more primitive entries, which are them linked with 'linguistic atoms', lexical items. Lexical items are provided to the syntax as primitive constituents, with all their properties (features) coming from the lexicon. 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 stream, but converted into *features* instead of morphemes or grammatical heads in syntax and then inserted inside adjacent lexical items.

Lexical features emerge from three distinct sources. One source is the language-specific lexicon, which stores information that is specific to a 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 in the form of feature implications. In this case, the redundancy rule states that the feature CAT:v implies the existence of feature COMP:V. When a lexical item is retrieved, its feature content is fetched from the language specific lexicon and processed through the redundancy rules. If there is a conflict, language-specific lexicon wins.

3.10 Argument structure

The notion of argument structure refers to the structure of thematic arguments and their predicates at their canonical LF-positions, the latter which are defined by means of theta role assignment and by tail-head dependencies.

The thematic role of ‘agent’ is assigned at LF by the small verb v to its specifier, so that a DP argument that occurs at this position will automatically receive the thematic role of ‘agent’. The parser-grammar does not see the interpretation, however, it only sees 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 the 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

whole verb class, and that feature should be associated with its feature structure by means of 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 (38). 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.

(38)

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

The syntactic representation of an argument structure raises nontrivial and interesting questions concerning the relationship between grammar and meaning. Consider a simple sentence such as *John dropped the ball* and its meaning, illustrated in Figure 8.

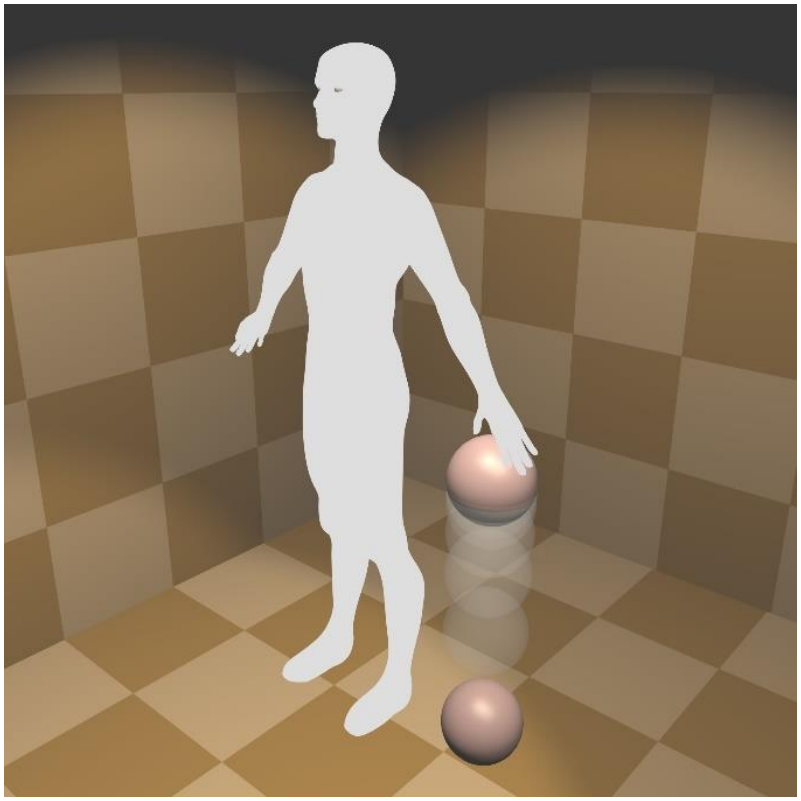


Figure 8. A non-discursive, perceptual or imagined representation of the meaning of the sentence *John dropped the ball*.

The canonical LF-representation for the sentence is mapped compositionally to the above representation in such a way that the innermost V-DP configuration represents the event of ball's falling, while the v-V adds the meaning that the event is originated by an agent or force. The subject then adds what that force is. This provides the idea that it is John that causes the ball to fall or is responsible for the event's occurrence Therefore, addition of elements to the LF-structure adds elements to the event (as imagined, perceived or hallucinated). Functional heads add independent components, such as aspect or tense, others, like v, make up unsaturated components 'x causes an event' that require a further object 'x'. Under this conception, the role of syntactic selection features is to allow the language faculty to create configurations that can be provided a legitimate interpretation of the type illustrated in Figure 8. On the other hand, they do not guarantee that an interpretation emerges: a sentence like *John dropped the democracy furiously* is as possible as *John dropped the ball by accident*.

3.11 Agree-1

Most languages exhibit an agreement phenomenon illustrated in the example (39).

(39) John [admire + *s* Mary]
3sg 3sg

The term "agreement" or "phi-agreement" refers to a phenomenon in which the gender, number or person features (collectively called phi-features or φ-features) of one element, typically a DP, covary with the same features on another element, typically a verb, as in (39). This covariation can be thought of as establishing a 'grammatical link' between the two elements, in the sense that one of the elements is registering the presence of the other. For example, if the subject were in the plural, the agreement suffix +*s* would disappear from the above sentence.

Not all lexical elements express or host overt phi-features, and those which do can be separated at least into three classes with respect to the type of agreement that they exhibit; and the agreement, when it is possible in one of the three forms, is sensitive to structural conditions and constraints. We have to capture these distinctions, and the rules and conditions.

The property of whether a lexical element (say a verb, noun or adjective) exhibits any agreement, even in principle, is determined by lexical feature ±VAL. A lexical item with −VAL does not exhibit phi-agreement. In English, conjunctions (*but*, *and*) and the complementizer (*that*) belong to this class, and are marked for −VAL. Those lexical items which exhibit agreement in principle have feature +VAL. There is variation with respect the distribution of this feature in the lexicon. In Finnish, the sentential negation *e-* inflects like a verb, thus it is marked for +VAL, whereas in English the negation behaves like a particle and has −VAL diacritic. We can see from the above example that finite verbs in English are marked for +VAL. The third person singular agreement suffix +*s* tells us that agreement takes place. A property of this kind of required in order

to trigger overt agreement operations inside syntax. It could be that something else than a simple lexical feature is at stake, but whatever that "something else" could be must be demonstrated by computer simulation.

Those heads which phi-agree can be further divided into two groups: those which exhibit full phi-agreement with an argument and those which exhibit only concord. Whether a lexical item exhibits full phi-agreement with an argument or not is determined by feature ±ARG. Negative marking −ARG creates concord: phi-agreement in which the element is not linked with a full argument, but agrees more passively; the positive marking +ARG forces the element to get linked with a full argument. This linking will be interpreted at LF-interface by means of *predication*: the predicate has +ARG and will, therefore, get linked with its argument. In the above example, the finite verb, or more specifically the finite tense head T/fin, has +ARG and establishes an agreement relation with a full argument, in this case the subject of the sentence *John*.

These features make room for a fourth possibility, a predicate that is linked with an argument, but which does not phi-agree. This group will create *control*, discussed in the next section. The four options are illustrated in Table 1.

Table 1.

Four agreement signatures depending on features ±VAL and ±ARG.

	−VAL	+VAL
−ARG	Lexical items which neither exhibit agreement nor require arguments (particles, such as *but, also, that, not*)	Lexical items which exhibit agreement but do not require linking with arguments (agreement by concord, e.g. *piccolo, pienet*)
+ARG	Lexical items which do not exhibit agreement but require linking with arguments (control constructions, such as *to leave, by leaving*)	Lexical items which exhibit agreement and linking with arguments (finite verbs, *admires*)

What the actual phi-features are and how they are interpreted semantically depends on language. But when an element exhibits phi-agreement with some other element we can always distinguish the two agree-partners on the basis of the role the agreement features play in the semantic interpretation. An argument DP has interpretable and lexical phi-features which are connected directly to the manner it refers to something in the real or imagined extralinguistic world. Thus, *Mary* refers to a third person singular individual; it does not refer to a plurality of things, for example. These features will be abbreviated as φ, but when we want to be more explicit we can also white PHI:NUM:SG for 'singular number' and PHI:PER:3 for 'third person', and so on. These features are lexically present at least in the head of the DP, the D-element. A predicate, on the other hand, that must be linked with an argument will have phi-features that 'reflect' those of an argument. To model this asymmetry, a predicate with +ARG will have *unvalued* phi-features, denote by symbols φ_ or PHI:NUM:_. The term "unvalued" refers to the fact that such features are specified for the type (e.g., number, person) but not for the value. The value is literally missing, and will be provide by the argument with which

the predicate is linked with. This is shown in (40), in which I will still ignore the verbal agreement suffix *-s* present in the input.

(40) Mary [admires John]

D N	T/fin	v	V	D N
↓	↓			
PHI:NUM:SG	PHI:NUM:_			
PHI:PER:3	PHI:PER:_			
PHI:DET:DEF	PHI:DET:_			
	+ARG, +VAL			

The operation that fills the unvalued slots is called Agree-1. It values the unvalued features, if any, on the basis of an argument with which the predicate is linked with, if any (41). Agree-1 is applied only to heads with +VAL, which is the reason this feature exists in the system.

(41) Mary [admires John]

PHI:NUM:SG		PHI:NUM:SG
PHI:PER:3	→	PHI:PER:3
PHI:DET:DEF		PHI:DET:DET
	Agree^{-1}	

As a consequence of valuation, unvalued features disappear. If no suitable argument is found which to you for feature valuation, unvalued features remain. This scenario will be discussed in the next section.

A head with +ARG has unvalued phi-features in its lexical entry, which corresponds with an ‘empty argument hole’ or an ‘argument placeholder’. We can imagine that such functional heads denote unsaturated predicates in the Fregean sense; heads of this type are by their constitution predicates. The functional motivation for Agree-1 is therefore to try to saturate the predicate from the sensory input by locating an argument. This assumption will also explain why predicates are semantically and syntactically relational and require the presence of at least one other element: they contain, as an intrinsic lexical property, a placeholder for such elements.

The above example shows that some predicates arrive to the language comprehension system with overt agreement information. The third person suffix *+s* already by itself signals the fact that the argument slot of *admires* should be (or is) linked with a third person argument. The pro-drop phenomenon, furthermore, shows that this holds quite literally: in many languages with sufficiently rich agreement no overt phrasal argument is required. Example (42) comes from Italian.

(42) adoro Luisa.
admire.1sg Luisa
'I admire Luisa.'

Inflectional φ-features of predicates, like any inflectional features such as case, are extracted from the input and embedded as morphosyntactic features to the corresponding lexical items (43).

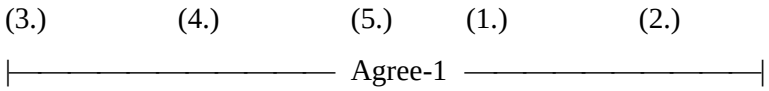
(43) John admire + s Mary.
↓ ↓ ↓
[John [admire Mary]]
{…3sg…}

This means that *admires* (or rather T/fin) will have both unsaturated phi-features due to +ARG and saturated phi-features as it arrives to syntax on the basis of the sensory input. Unsaturated features will still require valuation, which triggers Agree-1 (if the predicate is marked for +VAL). The existing valued phi-features, if any, impose two further consequences to the operation. First, Agree-1 must check that if the head already has valued phi-features, no phi-feature conflict arises when it finds a full argument. Thus, a sentence such as **Mary admire John* will be recognized as ungrammatical by Agree-1. Second, we allow Agree-1 to examine also the valued phi-features inside the head if (and only if) no overt phrasal argument is found. The latter mechanism will create the pro-drop signature. We thus interpret a valued phi-set φ inside a head as if it were a 'truncated pronominal element', call it pro. Sentence (44) should therefore be interpreted literally as 'I.admire Luisa'.

(44) adoro Luisa.
admire.1sg Luisa
admire.pro Luisa
'I admire Luisa.'

Agree-1 is limited to local domain, with possible language-specific variation (not yet modeled). It works by searching for DP-arguments inside a local domain such that the D contains valued phi-features that it can use to value its φ_. The local domain is defined by (i) its sister and specifiers inside its sister; (ii) its own specifiers; and (iii) the possible truncated pro-element inside the head itself, in this order. The first suitable element that is found is selected. This is illustrated in (45), with the ordinal numbers (1.-5.) showing the order in which the structure is explored. Here I assume, for generality, that HP and GP contain several specifiers, but this is not the norm.

(45) [$_{HP}$ Spec [Spec [H [$_{GP}$ Spec [Spec [G XP]]]]]]
DP DP pro DP DP



The specifiers and the head of a projection will be called its *edge*. Agree-1 occurs after head reconstruction, adjunct reconstruction and A-bar/A reconstruction and right before the structure is passed on to the LF-interface; thus, it targets elements at their canonical positions. The structure corresponds roughly to the d-structure in the standard theory. The reason nominative argument agrees with the finite verb can now be derived because the nominative case will guide the argument into the SpecvP position just below T/fin, and this position is prioritized by Agree-1 (45).

3.12 Antecedents and control

The assumptions specified in the previous section leave room for a situation in which an unvalued phi-feature or a whole phi-set arrives to the LF-interface unvalued. This outcome may occur for two reasons. One possible reason for the presence of unvalued phi-features at the LF-interface is if Agree-1 is not successful and is not able to locate a suitable DP-argument from the vicinity of the predicate. Another scenario occurs if the predicate has unvalued phi-features (is marked for +ARG) but cannot value these features at all (−VAL). In both cases an unvalued feature or features trigger(s) an *LF-recovery operation* that attempts to find a suitable argument by searching for an *antecedent*. An antecedent is located by establishing an upward path from the triggering feature/head to the antecedent. This can be illustrated by using an English infinitival verbs that do not agree and are therefore marked (by assumption, for the sake of the example) for −VAL. The infinitival verb cannot locate an argument, whether it implements Agree-1 or not, and therefore satisfies its unvalued features by finding an antecedent by means of an LF-recovery, marked by = symbol in the example. The result is an interpretation in which John is both the agent of wanting and the agent of leaving:

(46) John wants to leave$_{\varphi_ = \text{John}}$
‘John$_1$ wants: John$_1$ to leave.’

The upward path (or ‘upstream walk’ in the implementation, when looked as a step-by-step procedure) is defined by an operation which looks at the sister of the head H, evaluates whether it constitutes a potential antecedent and, if not, repeats the operation at the mother of H. If LF-recovery finds no antecedent, the argument is interpreted as generic, corresponding to ‘one’ (47).

(47) To leave$_{\varphi_=\text{gen(‘one’)}}$ now would be a big mistake.

It may also happen that Agree-1 succeeds only partially, leaving some phi-features unvalued. This is the case with the third-person agreement in Finnish which, unlike first or second person, triggers LF-recovery. Anders Holmberg has argued that this happens because the Finnish third person agreement suffix, in contrast

to say Italian third person suffix, cannot value the unvalued D_ feature. Assume so. Then D_ triggers LF-recovery at LF, as shown by (48).

(48) Pekka sanoi että nukkuu hyvin.
Pekka said that sleep.3sg$_{D_=Pekka}$ well
'Pekka said that he (=Pekka) sleeps well.'

This illustrates a situation in which Agree-1 takes place by fails or perhaps succeeds only partially. Overall, this architecture suggests that unsaturated features of predicates must be saturated and, if neither Agree-1 nor LF-recovery succeeds, they are saturated as being generic as a last resort. Perhaps grammatical heads marked for +ARG are by their very semantic constitution such as they require an argument to be interpretable, and the phi-features provide 'minimal' referential properties required to make then intelligible for the conceptual-intentional system.

4 Formalization and implementation

4.1 General organization

The model was implemented and formalized as a Python 3.x program. It contains three separate components. The first component is a *main script* responsible for performing and managing testing. It reads an input corpus containing test sentences and other input files, such as those containing lexical information, prepares the parser (with some language and/or other environmental variables), runs the whole test corpus with the parser, and processes and stores the results. The architecture is illustrated in Figure 12.

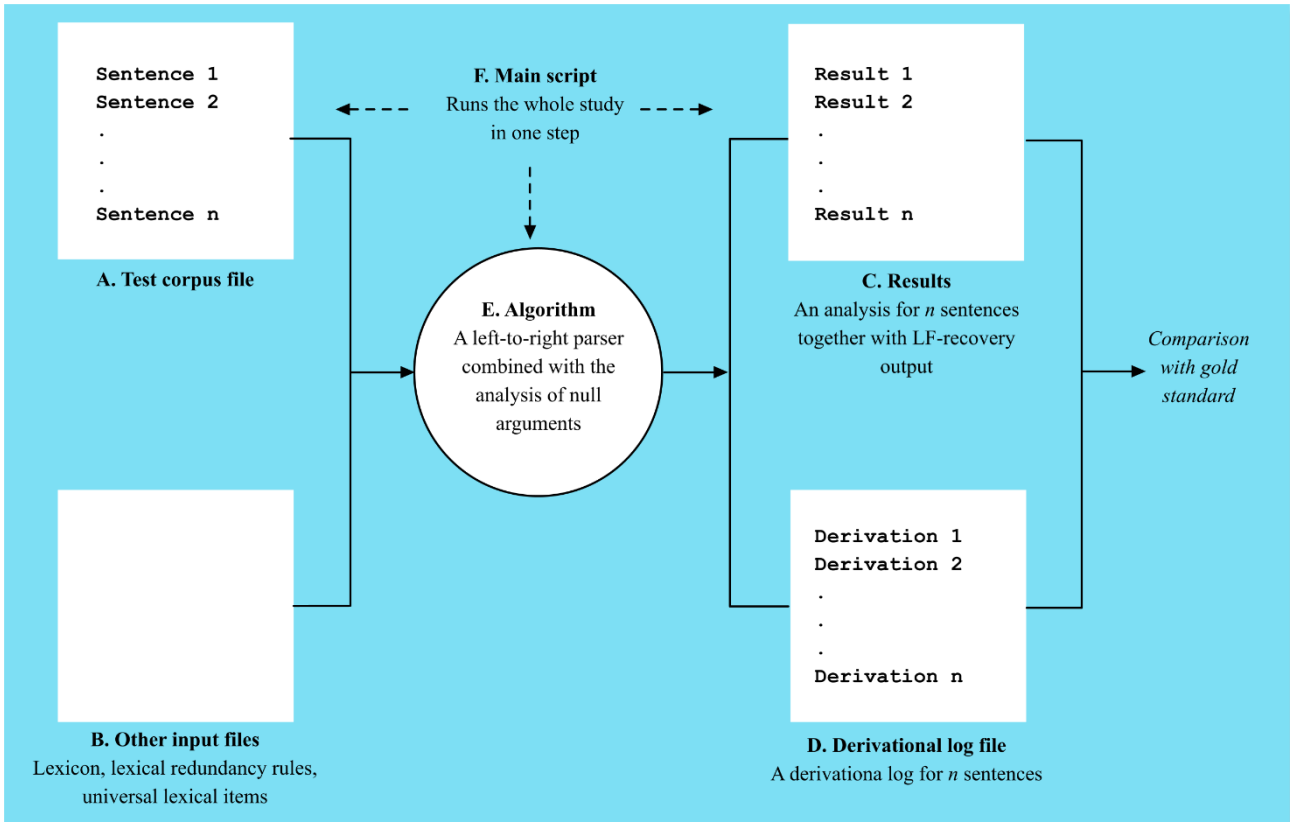


Figure 12. Relationships between the input, main script, linear phase parser and the output.

Any complete study is run by launching the main script once, which provides a mapping between the input files and output files, and which are then all stored as the "raw data" associated with each study. The whole study is processed by launching the main script once: it processes everything and stops when everything is done. The user cannot and should not interfere in the process during the execution, so that it will remain deterministic and unambiguous. The second component is the language comprehension module itself, which receives one sentence as input and produces a set of phrase structures and semantic interpretations as output. That component contains the empirical theory, formalization of the assumptions introduced nontechnically in Section 2. Finally, the program contains support functions, such as logging, printing, and formatting of the results, reporting of various program-internal matters, and others. These are not part of the empirical theory.

The code exists in separate Python text files. The required files are contained in the master folder and in a few subfolders. The individual modules, containing the program code, are ordinary .py text files that are all located in the master folder. The main script is called *parse.py*. It currently takes no input parameters; rather, the input and output files are specified inside the code itself (again, which is just an ordinary text file). This reflects the way the program is used in a typical scenario, namely by executing numerous trial and error runs with the same parameters. It would be easy to add the input parameters, though, if needed. The main script then calls the parser, which uses several other modules, each in its own file that contains definitions for just one module or class. All these files, and thus all program modules, are currently in the same master folder

where the main script is. But because individual studies are associated with specific input files (test corpus, lexical files), these are organized into the *language data working directory* subfolder, which contains a further subfolder for each study, published, submitted or in preparation. This allows one to separate different studies from each other.

To properly replicate a published study also the code is required. The code versions used in connection with any given study are stored in separate folders in the cloud together with the rest of the material associated with that study, including all manuscript versions, correspondence, reviews, figures, and such. This material can be accessed by the author and is provided if needed. In addition, however, the program branches are stored at Github, but because some of the datafiles are very large the ultimate storage medium for any individual study must be the cloud that is able to store huge volumes of data. It should be possible to replicate any study by using the resources available at Github.

The program version numbering follows published studies. Version 1.0 was associated with the first published study, 2.0 with the second, and so on. Pre-publication versions are labeled in a similar way, thus the penultimate version of the model associated with the published study number 1 is 0.99. Version number 5.x, the version associated with the present document, refers to a version have been implemented after study 5. The published studies are the following: 0 = the first versions of the algorithm described in the first versions of this document (Brattico 2019a); 1 = pied-piping and operator movement (Brattico and Chesi 2020); 2 = free word order in Finnish (Brattico 2019b); 3 = control and null arguments (Brattico 2020b); 4 = head movement reconstruction and Finnish long head movement (Brattico 2020c); 5 = first study of clitics and incorporation (Brattico 2020a); 6 = convergence version taking care of all the data (in preparation).

4.2 Main script (parse.py)

The main script is divided into 4 blocks. It begins by importing other code modules that are required in its internal computations (block 1), which is followed by definitions for input-output files and their names (block 2). The names and folders provided in block 2 are important, because they define where the input files are located and what names will be used when producing the output. The user defines the folder and the name of the input corpus, and the main script provides default names for everything else. These can be changed into whatever conventions are adopted. Block 3 contains various preparations, such as logging functions, current time, prints information concerning the execution and parameters to the console and reads the test corpus file, based on the names provided inside block 2. Block 4 contains the actual parsing loop which sends each sentence one by one to the parser and then stores the results into the results file, as defined inside block 2. The structure of the main script is summarized in Figure 13.

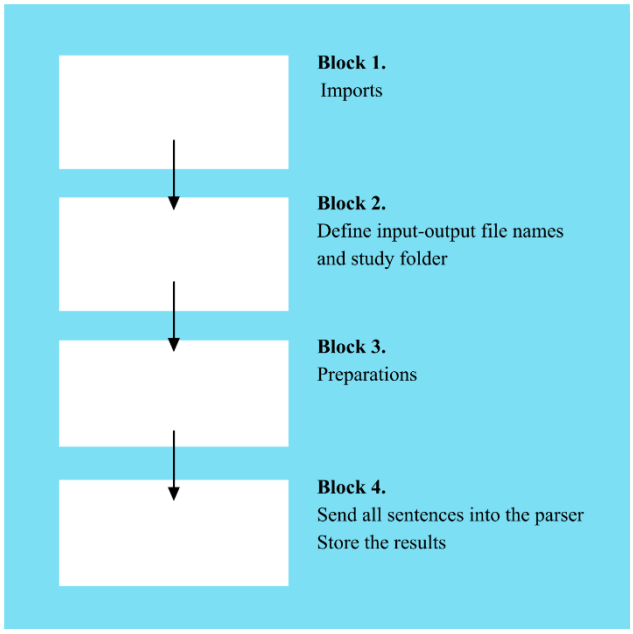


Figure 13. Structure of the main script, which consists of four blocks that are executed in a linear order. The input-output files in block 2 are changed to match the study. The nature of the output – i.e. what type of information we wish to store – is defined inside block 4.

4.3 Structure of the input files

4.3.1 Test corpus file (any name)

The test corpus file name is provided in the main script and it can be anything. Certain conventions must be followed when preparing the file. Each sentence is a linear list of phonological words, separated by space from each other and following by next line (return, or \n), which ends the sentence. The words that appear in the input sentences must be found from the lexicon exactly in the form they are provided in the input file, which means that the user must normalize the input. For example, do not use several forms (e.g. *admire* vs. *Admire*) for the same word, do not end the sentence with punctuation, and so on. Depending on the research agenda you might want to consider using a fully disambiguated lexicon. While it would be easy to normalize the input during execution, that would introduce a potential source of error that I think we should avoid.

Special symbols are used to render to output more readable and to help testing. Symbol # in the beginning of the line is read as a comment and is ignored. Symbol & is also read as a comment, but with the exception that it will appear in the results file as well. This allows the user to leave structural comments into the results file that would otherwise get populated with raw data only. If a line is prefaced with %, the main script will process only that sentence. This functionality is used if you want to examine the processing of only one sentence in detail. If you want to examine group of sentences, they should all be prefaced with + symbol. The rest of the sentences are then ignored. Finally, command =STOP= at the beginning of a line will cause the processing to stop at that point, allowing the user to process only *n* first sentences. If we want to process

sentences from *n*th sentence, we can define that inside the main script block 3. Figure 14 is a screen capture from one test corpus file to illustrate what it looks like.

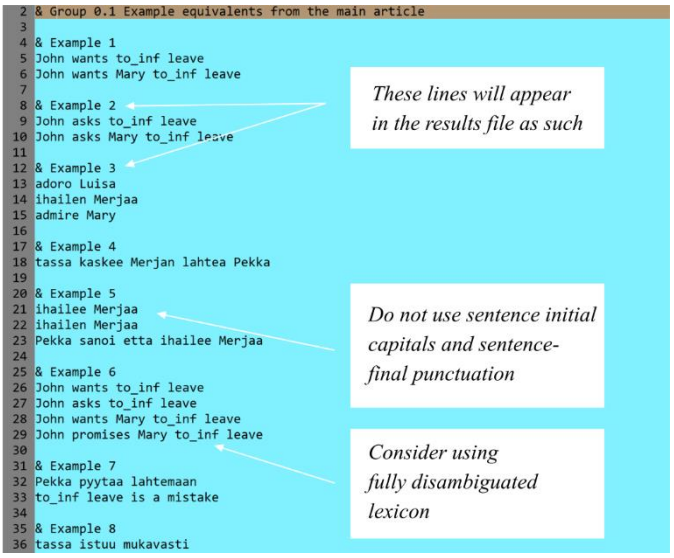


Figure 14. Screen capture of a test corpus file.

4.3.2 Lexical files (lexicon.txt, ug_morphemes.txt, redundancy_rules.txt)

The main script uses three lexical resource files that are by default called *lexicon.txt*, *redundancy_rules.txt* and *ug_morphemes.txt*. These names are defined in the main script and can be changed by changing the code there. The first contains language specific lexical items, the second a list of universal redundancy rules and the last a list of universal morphemes. Figure 15 illustrates the language-specific lexicon.

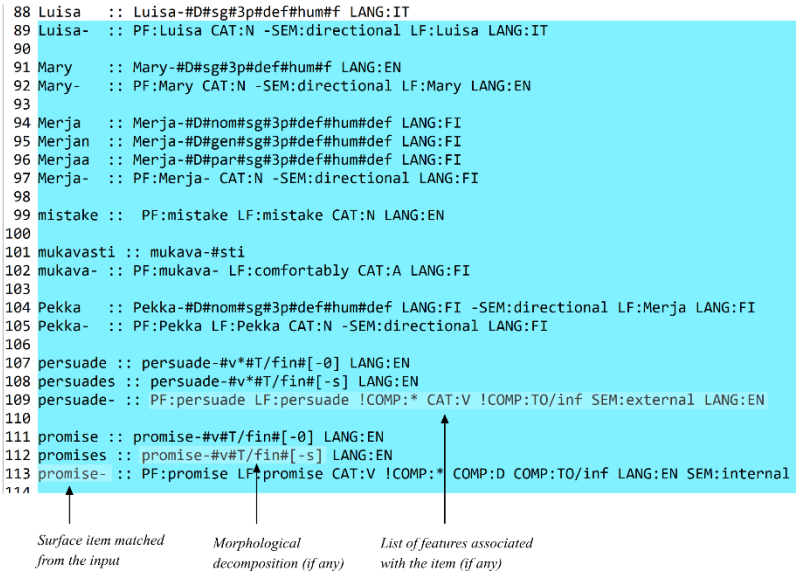


Figure 15. Structure of the lexical file (*lexicon.txt*)

Each line in the lexicon file begins with the surface entry that is matched in the input. This is followed by symbol :: which separates the surface entry from the definition of the lexical item itself. If the surface entry has morphological decomposition, it follows the surface entry and is given in the format '*m#m#m#…#m*' where each item *m* must be found from the lexicon. Symbol # represents morpheme boundary. The individual constituents are thus separated by symbol # which defines the notion of morphological decomposition; do not use this symbol anywhere else in the lexical files. If the element designates a primitive (terminal) lexical item, it has no decomposition; instead, the entry is followed by a list of *lexical features*. Each lexical feature will be inserted as such inside that lexical item, in the *set* constituting that item, when it is streamed into syntax. There is no limit on what these features can be, but narrow syntax and semantic interpretation will obviously register only a finite number of features.

A lexical feature is a string, a "formal pattern" or ultimately a neuronal activation pattern. They do not have further structure and are processed by first-order Markovian operations. The way any given feature reacts inside syntax and semantics is defined by the computations that process these lexical items and the "patterns" in them. As can be seen from the above screen capture, most features have a 'type:value' structure. Figure 16 contains a summary of the most important lexical features that the syntax and semantics reacts in the current model.

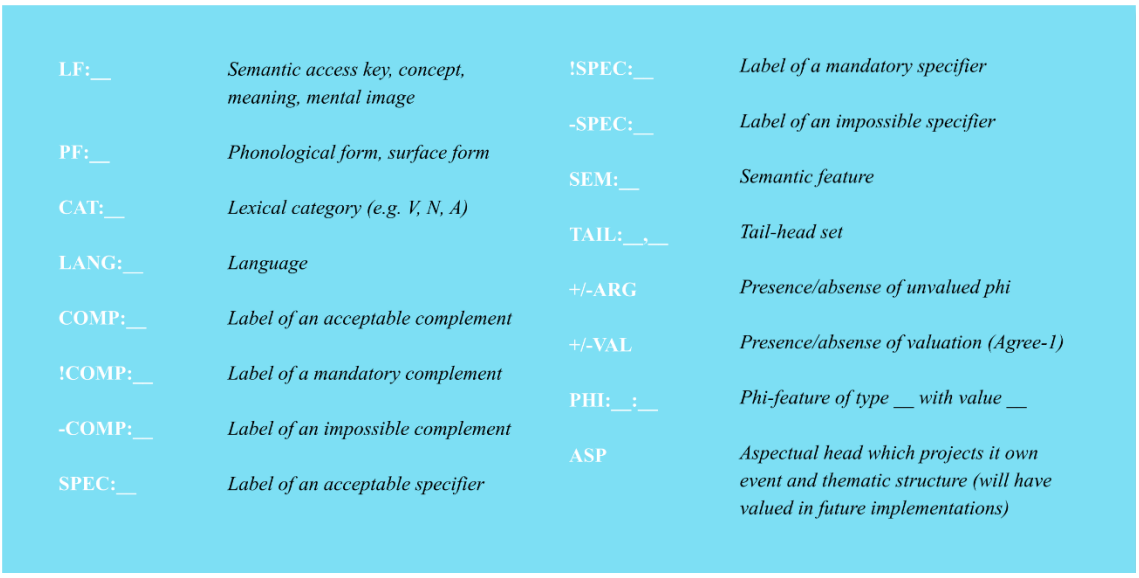


Figure 16. Selection of lexical features

The file *ug_morphemes.txt* is structured in the same way but contains universal morphemes such as T and v. An important universal feature category is constituted by *inflectional features* such as case features and phi-features. An inflectional feature is designated by the fact that its morphemic decomposition is replaced with symbol –. They are otherwise defined as any other lexical item, namely as a set of features. These features are inserted inside full lexical items during morphological decomposition and streaming of the input into

syntax. The word *admires* is processed so that the third person singular features (PHI:NUM:SG, PHI:PER:3) are inserted inside T/fin.

4.3.3 Lexical redundancy rules

Lexical redundancy rules are provided in the file *redundancy_rules.txt* and is used to define default properties of lexical items unless otherwise specified in the language-specific lexicon. Redundancy rules are provided in the form of an implication '$\{f_0, \ldots, f_n\} \rightarrow \{g_1, \ldots, g_n\}$' in which the presence of a triggering features $\{f_0, \ldots, f_n\}$ in a lexical item will populate features $g_1, \ldots, g_n$ inside the same lexical item. It is illustrated in Figure 17 which is a screen capture from a redundancy rule file.

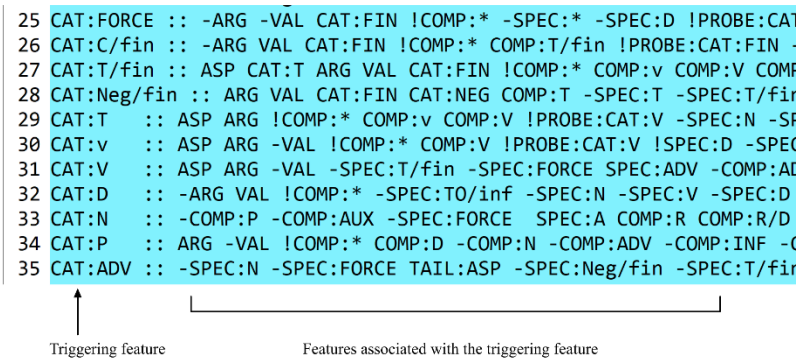


Figure 17. Lexical redundancy rules, as a screen capture from the *redundancy_rules.txt* file.

The antecedent features are written to the left side of the :: symbol, and the result features to the right. Both feature lists are provided by separating each feature (string) by whitespace. In Figure 17, all antecedent features are single features.

The lexical resources are processed so that the language-specific sets are created first, followed by the application of the lexical redundancy rules. If a lexical redundancy rule conflicts with a language-specific lexical feature, the latter will override the former. Thus, lexical redundancy rules define "default" features associated with any given triggering feature. They are universal and define a 'lexical mini grammar'; there are no language-specific lexical redundancy rules in this implementation.

4.4 Structure of the output files

4.4.1 Results

The name and location of the results output file is determined in the main script. The default name is made up by combining the test corpus name together with "_results". Each time the main script is run, the default results file is overridden. Therefore, once you get results that are plausible, it is a useful to rename the results file to safe it and compare with new output. The file begins with time stamps together with locations of the input files, followed by a grammatical analysis and other information concerning each example in the test

corpus, with each provided with a numeral identifier. What type of information is visible depends on the aims of the study. The example in Figure 19 shows one grammatical analysis together with the output of LF-recovery.

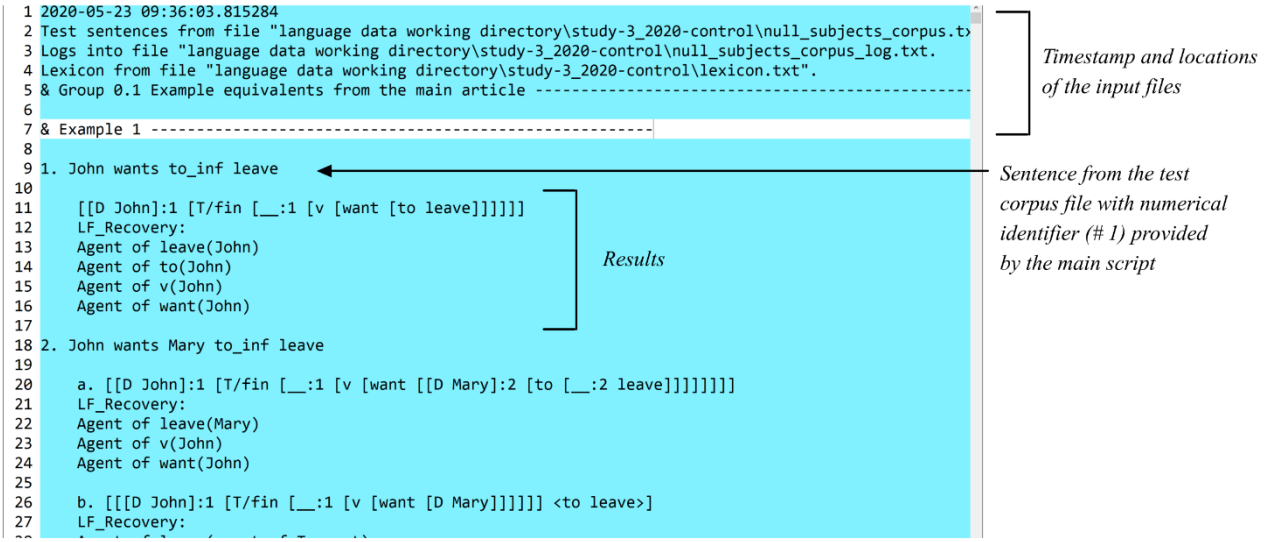


Figure 19. Screen capture from a results file.

4.4.2 The log file

The derivational log file, created by default by adding "_log" into the name of the test corpus file, contains a more detailed report of the computational steps consumed in processing each sentence in the test corpus. The log file uses the same numerical identifiers as the results file. In order to locate the derivation for sentence number 1, for example, you would search for string "# 1" from the log file. What type of information is reported in the log file can be decided freely. By default, however, the log file contains information about (1) the processing and morphological decomposition of the phonological words in the input, (2) application of the ranking principles leading into Merge-1, and (3) transfer operation applied to the final structure when no more input words are analyzed. Intermediate left branch transfer operations are not reported in detail. The beginning of a log file is illustrated in Figure 20, containing examples of operations (1-2).

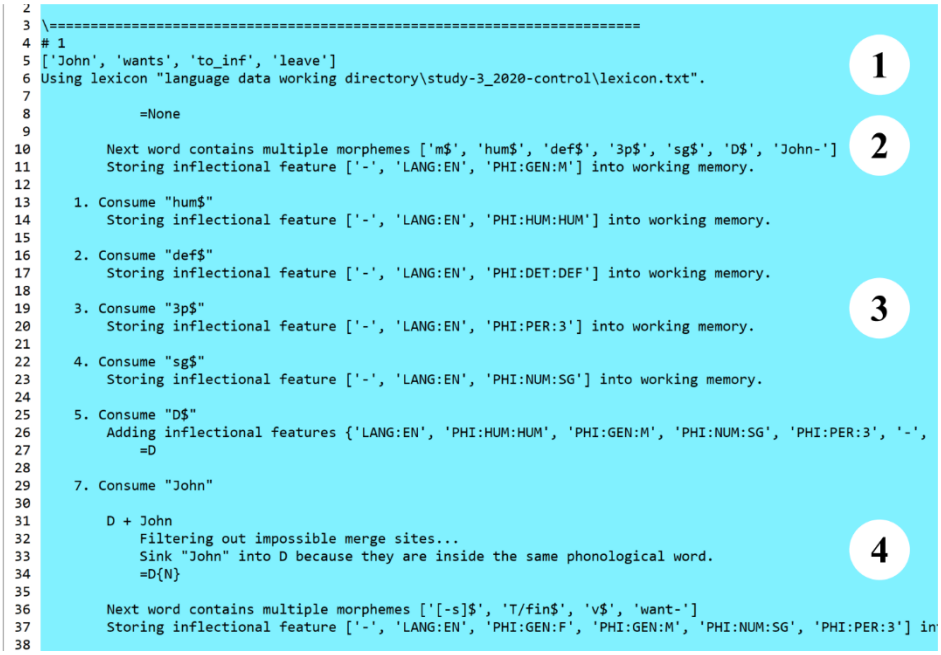


Figure 20. Screen capture from the log file. (1) Input sentence and its numerical identifier, together with information concerning the lexicon file used in the processing. (2) The next word is multimorphemic and is decomposed into a list of elements, here both inflectional features and grammatical heads (D, John). (3) Inflectional features are stored into a working memory and are then added to the adjacent D-morpheme. (4) Here we consume the item *John* = N that can be merged-1 with D. The sentence “Sink ‘John’ into D because they are inside the same phonological word” means that N is inserted inside D, as seen in the output D{N}.

Figure 21 illustrates the log file of transfer operations.

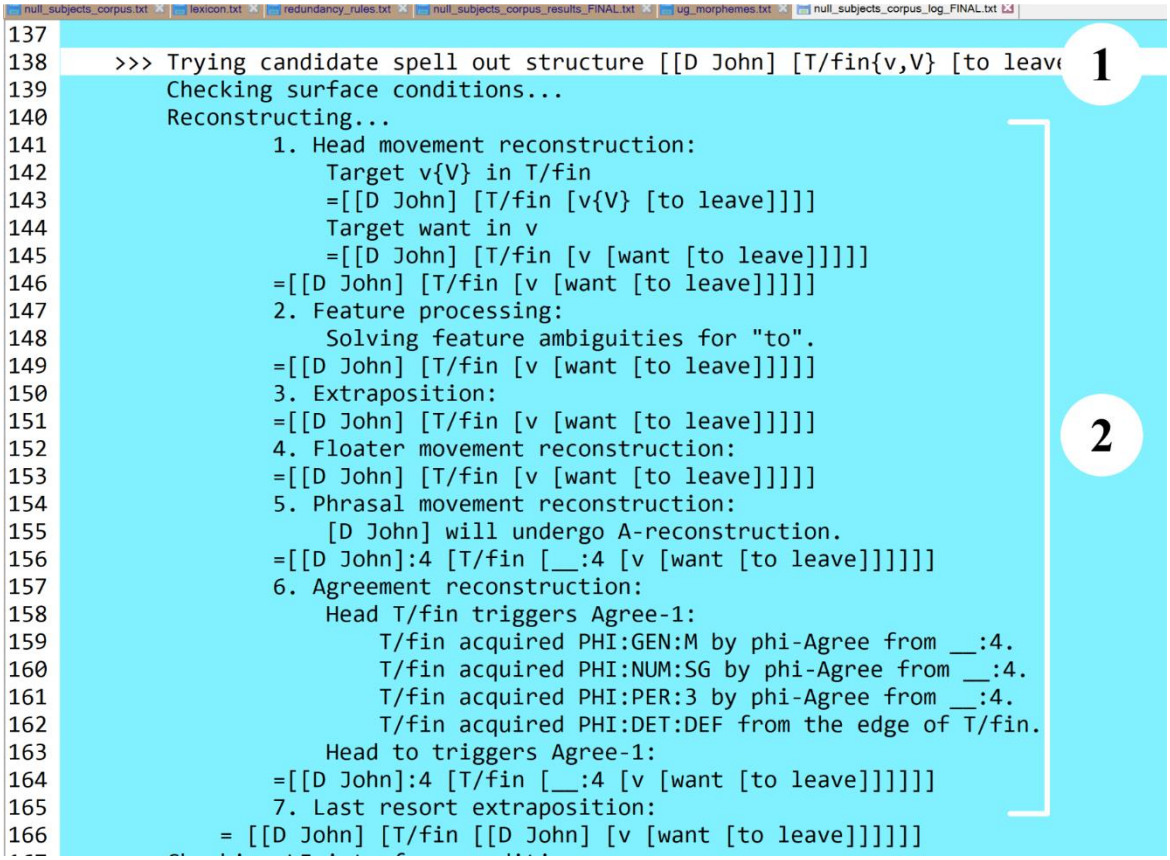


Figure 21. Transfer in the log file. The first line (1) states the spellout structure to be transferred. The transfer operations then following the order of their execution (2). The end result of each step is prefixed with =.

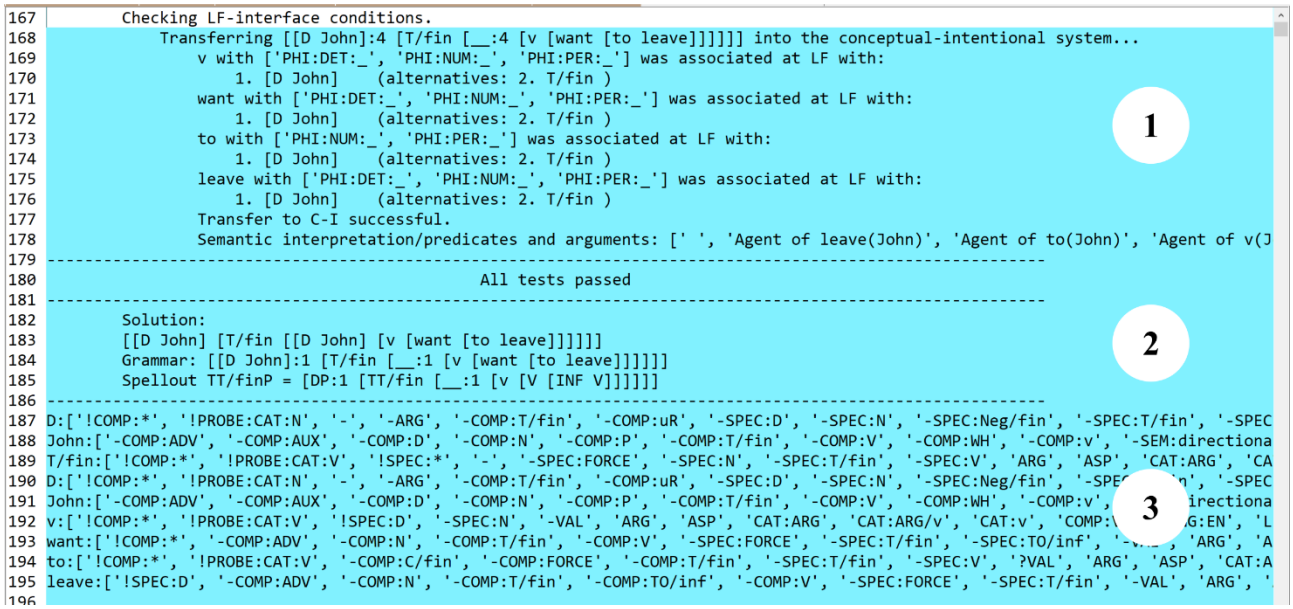


Figure 22. Post-transfer operations in the log file: (1) LF-interface legibility check which establishes whether the structure can be interpreted semantically and provides information concerning LF-recovery; (2) output, if LF-legibility tests passed; (3) feature content of each element in the output.

4.4.3 Saved vocabulary

Each time a study is run, the program takes a snapshot of the surface vocabulary (lexicon) as it stands after all processing has been done (after each sentence has been processed) and saves it into a separate text file with the suffix *_saved_vocabulary.txt*. The reason is because the ultimate lexicon used in each study is synthesized from three sources (language-specific lexicon, universal morphemes and lexical redundancy rules) and thus involves computations and assumptions whose output the user might want to verify independently. Notice that the complete feature content of each terminal element that occurs in any output solution is stored into the log file together with the solution (Section 4.4.2) and does not appear in this file.

4.4.4 Images of the phrase structure trees

The visualization module, which is an auxiliary tool, is constantly under development and improvement. The algorithm stores the parsing output in phrase structure images (PNG format) if the user activates the corresponding functionality. The functionality can currently be activated by setting *image_output = True* in the block 3 inside the main script. The variable *stop_after_each_image*, when set *True*, will halt the processing after each phrase structure tree is presented on a separate window, and continues only if the user closes that window. If the variable is set to *False*, the program will still produce the windows and phrase structure tree but closes them immediately. This slows down the processing, however, and is not implemented in optional manner. The images are stored into folder *phrase_structure_images* under the working directory (if the folder does not exist, the program will create it). The files are named according to the numbering generated during processing, and thus they match with the number identifier found from results files and derivational log files. Figure 23 illustrates the phrase structure representation generated for a simple transitive clause in English.

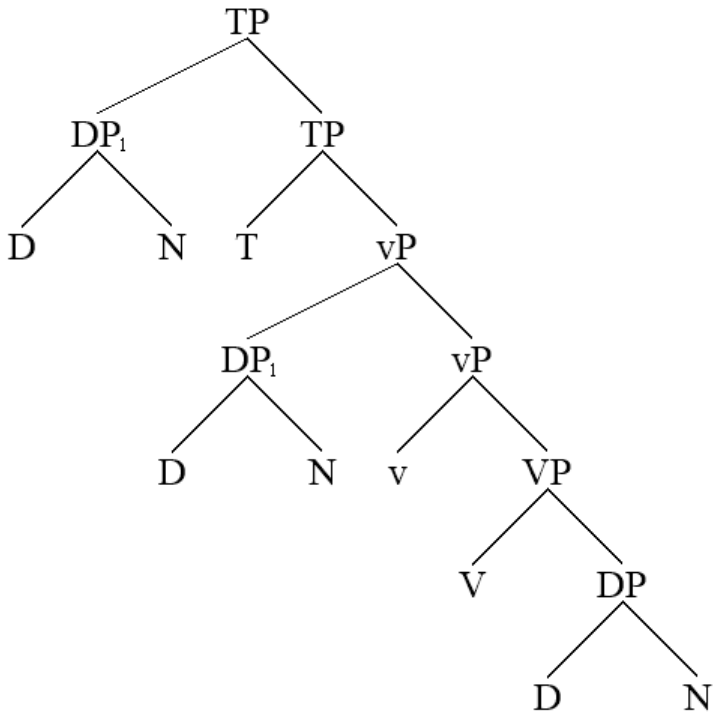


Figure 23. A simple phrase structure image generated by the algorithm for a simple transitive clause *John admires Mary*.

The images are relatively austere when compared with the information present in the derivational log files. This is because they are meant to be used as illustrations and also as raw templates that the user can use to craft more informative images. The lexical category labels are drawn from the list of major categories defined at the beginning of the *phrase_structure.py* module. If the label of an element is not recognized, it will appear as X (and XP for phrases).

4.4.5 How to add your own log entries

The user can add your own logging entries directly inside the code. The command is log('information to be logged').

4.5 Linear phase parser (linear_phase_parser.py)

4.5.1 Definition for function *parse(list)*

The linear phase (LP) parser module (linear_phase_parser.py) defines the behavior of the parser. When main script wants to parse a sentence, it sends the sentence to this module as a list of words. The parsing function is *parse(list)*. The parser function prepares the parser by setting a host of variables (mostly having to do with logging and other support functions), and the calls for the recursive parser function *_first_pass_parse* with three arguments *current structure*, *list* and *index*, with *current structure* being empty and *index* being 0 in the

beginning. This function processes the whole list, creates a recursive parsing tree, and provides the output. I do not think that the real human parser is recursive, but full recursion is required to examine competence, so that this module should be thought of as an idealization or generalization of the real parser.

4.5.2 Definition for function *_first_pass_parse(current structure, list, index)*

The recursive parsing function takes the currently constructed phrase structure α, a linearly ordered 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, α will be transferred to LF and evaluated. Transfer normalizes the phrase structure by reverse-engineering movement and agreement and performs LF-legibility tests to check that the output is semantically interpretable. If the test passes, the result will be accepted. Executive control is then passed to the conceptual-intentional system that is not modeled here. If the user wants to explore all solutions, then the algorithm will backtrack to search for further solutions.

Suppose word w was consumed. To retrieve properties of w, lexicon will be accessed. This operation corresponds to a stage in language comprehension in which an incoming sensory-based item is matched with a lexical entry in the surface vocabulary. If w is ambiguous, all corresponding lexical items will be returned as a list [w_1, w_2, . . ., w_n] that will be explored in some order. The ordered list will be added to the recursive loop as an additional layer. The ordering is arbitrary in the current interpretation but not so in realistic parsing.

Next a word from this list, say w_1, will be subjected to morphological parsing. Suppose the word is *pudo-t-i-n-ko-han* ‘fall-cau-past-1sg-Q-hAn’. Morphological parser will return a new list of words that contains the individual morphemes that were part of w_1, in reverse order, together with the lexical item corresponding with the first item in the new list. The new list therefore contains a ‘morphological decomposition’ of the word. 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 and hence also adjacent morpheme when it is being consumed in the reversed order. 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. A complex phonological word such as *pudo-t-i-n-ko-han* will enter syntax in the form (49). Notice that the order of the morphemes is reversed. The linearly ordered sequence that enters syntax is called the *lexical input stream*. The lexical input stream does not contain phonological words, but lexical items and features.

(49) pudo-t-i-n-ko-han (input word)
fall-cau-past-1sg-Q-hAn → (decomposition)
hAn * Q * 1sg * T * v * V (reverse order into syntax, lexical input stream)

The information that all these items were part of the same word is retained and passed on to syntax, which prevents it from considering merge solutions that are not compatible with their word-internal status.

Suppose morpheme *hAn* was consumed. The morpheme must be merged to the existing phrase structure into some position. Merge sites that can be ruled out as impossible by using local information are filtered out. The remaining sites are then ranked (Section 4.5.3). Each site from the ranking is 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 embedded into the previous word to create a complex head [$_{\beta}$∅, α]. Thus, the morphemes in (49) are first processed in this way and create a complex head [$_{hAn}$∅, [$_{Q}$∅, [$_{T}$∅, [$_{v}$∅, V]]]]. If α was not inside the same word as the previous item, it will be merged countercyclically to the existing phrase structure, and the parsing function is called recursively. If a ranked list has been exhausted without a legitimate solution, the algorithm will backtrack to an earlier step.

The complex linearly structured head [$_{hAn}$∅, [$_{Q}$∅, [$_{T}$∅, [$_{v}$∅, V]]]] corresponds to a situation in the more standard bottom up theories in which all of the morphemes have been 'snowballed' out of the structure to the highest node. In the standard theories, the heads would have been adjoined with each other. This solution did not produce elegant results here, because any constituent with both left and right constituents is automatically regarded as a standard complex constituent and not a head. This would make any complex head a phrasal specifier. Various ad hoc strategies are used in the standard theories to prevent this outcome, but these devices are all questionable. The structure of *_first_pass_parse* is illustrated in Figure 18.

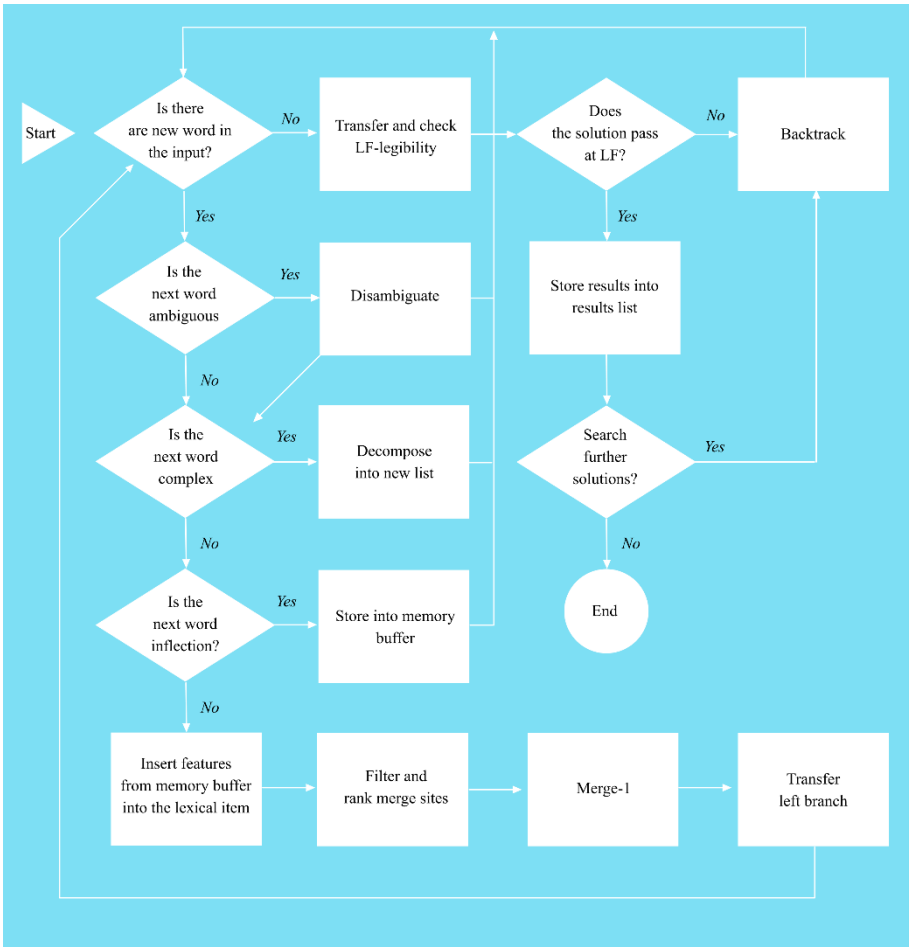


Figure 18. Flowchart of the recursive parsing algorithm *_first_pass_parse()*.

4.5.3 Definitions for function *filter()*

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. Let us assume that α is the new element to be merged and β is the site of merge.

(50) *Conditions for filtering*

Condition 1. If the new morpheme α was inside the same word as the previous one, all other solutions except 'merge to the complement' (bottom of β) will be filtered out. Thus, v can only be complemented to T if they were originally inside the same word in the input. The rationale for this rule is obvious.

Condition 2. Primitive sites that do not accept any type of complements are filtered out. Justification is trivial.

Condition 3. If β is not primitive, it will be reconstructed (Section 4.6.6) and tested for LF-legibility: the legibility of β is always tested upon [β, α], and if β does not pass the solution is either rejected or ranked lower, depending on the type and severity of the violation. The filter function does not implement ranking, only filtering. Notice that the legibility test for β presupposes that β is normalized and goes through transfer

first. Complete rejection occurs if (i) LF-legibility fails, (ii) α is not adjoinable and (iii) the probe-goal test, head integrity test and the criterial feature integrity test (Section 4.10) all failed (if any of them succeeds, filtering is blocked).
Condition 4. If the solution breaks existing words, the solution is rejected. This test is not applied if α can be adjoined.

Condition 2 requires a comment. The reason [β, α] can be filtered under some conditions is because some legibility errors in β cannot be fixed in the derivation's future. Hence the solution can be rejected. Condition (iii) mean that the failure of the three tests in conjunction is so severe that the problem can be deemed as unrepairable, but it might be that the failure of each test individually is sufficient (the matter requires further examination and is possibly a mistake). The issue impacts computational complexity but not grammaticality.

4.5.4 Definition for function *ranking()*

The ranking function weights each solution based on various criteria. 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. The ordering does not affect solutions found by the algorithm, only efficiency and the first solution returned by the algorithm. Suppose the new element is α and the site to be tested is β, so that we are testing for [β α]. The criteria used in the current version are the following:

(51) *Ranking criteria*

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

4.6 Basic grammatical notions (phrase_structure.py)

4.6.1 Introduction

The class PhraseStructure (defined in *phrase_structure.py*) defines the phrase structure objects that are manipulated at each stage of the processing pipeline. The code is partitioned into six blocks: basic structure building functions (block 1), definitions for basic grammatical relations (block 2), definitions for nonlocal dependencies (block 3), functions which return properties inside lexical item (grammatical head)(block 4), unclassified (block 5) and support functions (block 6).

Phrase structure is assumed to be a generalization from linearization. The binary unit [α, β] represents a linearization or sensorimotoric programming of α and β. Recursion allows either α or β to be substituted with another similar unit, where linearization of [α, β] is 'interrupted' and α or β is linearized first.

4.6.2 Definition for cyclic Merge: *_init__(α,β)*

Simple Merge takes two constituents α, β and yields [α, β], α being the left constituent, β the right constituent. It is implemented in the class constructor function (__init__), which takes α and β as arguments and return a new constituent.

4.6.3 Definition for countercyclic Merge: *merge(α, β, direction)*

Countercyclic Merge-1 (*merge*(*α, β, direction*)) 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]. The latter operation uses cyclic merge (class __init__ function). 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. Both Merge to the right and left, and both countercyclically and by extending the structure, are allowed. The range of options is compensated by the restricted conditions under which each operation is allowed.

4.6.4 Definition for *remove*

An inverse of countercyclic Merge-1 is remove *(α.remove())*, which removes constituent α from the phrase structure and repairs the hole. 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.

4.6.5 Definition for *head*

Labeling algorithm was elucidated earlier. It is based on (52).

(52) Labeling

Suppose α is a complex phrase. Then

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

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

c. if the right constituent is not an adjunct, apply (15) recursively to it; otherwise,

d. apply (15) to the left constituent (whether adjunct or not).

4.6.6 Sister and geometrical sister: *sister*, *geometrical_sister*

A *sister* of α (defined by *sister(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 the mirror case [α [β, ⟨γ⟩]] the sister of β is α. Right adjuncts are invisible for sisterhood. A separate relation of *geometrical sisterhood* takes them into account (defined by *geometrical_sister(α)*).

4.6.7 Complement (*complement*)

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 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.

4.6.8 Specifiers and edges (*edge*)

We define the notion of *local specifier* or *local edge* so that it constitutes the left phrasal sister of the node immediately dominating α. If no such local left phrase exists inside αP, then the φ-features inside α, if any, can constitute a 'pronominal specifier' if the φ-set allows for the construction of an unambiguous pronoun. The relevance of local specifier is that it is the notion that enters into specifier selection.

(53) Local specifier

The local specifier for a left head α is β such that

a. [$_{\alpha P}$ β [α XP], β is complex or, if no such element exist,

b. a pro-element (if any) extracted from α.

A "pro-element" is an unambiguous φ-set associated with D. The specifier of a right head α is its complex left sister β (i.e. β in [β, α]). This condition is written as a separate condition inside the definition of specifier for clarity, but a uniform generalized notion should be possible.

A more general definition of *edge* or *generalized specifier* is also provided, which denotes any left phrase inside αP together with any pronominal material inside α if nothing else is found. Thus, the set of generalized specifiers of α contains all left phrases inside αP plus the pronominal element inside α, if any.

(54) Edge (generalized specifier)

β is a generalized specifier for head α if and only if either (a.) or (b.)

a. β occurs in a configuration [$_{\alpha P}$ β αP] and is a non-head;

b. β is extracted from α as a pro-element when nothing satisfies (a).

The elements are returned as a list that is ordered with respect to closeness to α (closest first). The definition is unusual in the sense that it return pro only if nothing else is found.

4.6.9 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, may 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 clarification. The clarification is that right-adjuncts 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. A *downstream* or *downward* path or walk follows this definition. *Upstream* path follows motherhood but skips over right adjuncts.

The *left branch* of [α, β] is α (in the case of [α, ⟨β⟩] the notion of left branch and downward branch coincide). 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).

4.6.10 Probe-goal: *probe(label, goal_feature)*

Suppose P is the probe head, G is the goal feature, and α is its (non-adjunct) sister in configuration [P, α]; then:

(55) Probe-goal

Under [P, α], G the goal feature, search for G from left constituents by going downwards inside α along its right edge (thus, 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). The fact that criterial feature search must be separate from the search of other types of features suggest that we are missing some piece of the whole puzzle.

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.

Consider again the case of C→T. When C searches for T, it cannot satisfy the probe-feature by going through another (embedded) C. If it did, lower T would be paired with two C-heads; this is semantically gibberish. Therefore, there is a "functional motivation" for the intervention condition.

4.6.11 Tail-head relation: *external_tail_head_test()*

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 mother-of relation. In the example (56), 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.

(56) $[_{AP}$ A^0 $[_{BP}[_{XP}$YP $X^0]$ $[B^0$ $[T^0$ 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 (and termination of search). Thus, if T has a tail feature (TAIL:F,G), but A has the feature F without G, the tail-head relation fails.

There is certain ambiguity in how the results of a tail-head relation are interpreted. One interpretation is that if full match is not found, the dependency fails. Another is that only the existence of partial match results in a failure; if nothing is matched, the test is still a success. The latter tests if an element is in a "wrong place," the former if it is in the "right place." Both type of tests are useful but in slightly different contexts. The former test is called *external tail-head test*, the latter *internal tail-head test*. The former (external test) is used when checking if a constituent with tail-head features must be moved to another position, in which it would satisfy the test. The latter (internal test) is applied when fitting a constituent with a case suffix and examining if it would appear under a wrong case assigner in that position; here only the presence of a wrong case assigners (tail-head feature) results in a failure, we don't care if nothing is matched. I think there is a better solution.

4.7 Transfer (transfer.py)

4.7.1 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. From an external point of view, it constitutes 'one' step; internally the operation consists of a sequence of steps. All movement inside α is implemented if and only if Merge-1(α, β)(Section 3.4). The operation targets α and follows a predetermined order: head movement reconstruction → adjunct movement reconstruction → A´/A-movement reconstruction → agreement reconstruction (i.e. Merge-1 → Move-1 → Agree-1).

4.7.2 Head movement reconstruction (head_movement.py)

Head movement reconstruction of [[$_{\alpha}\varnothing$, β], γ] can take place in one of two ways:

(57) Head movement reconstruction of [[$_{\alpha}\varnothing$, β], γ] can follow (a.) or (b.)

a. [[α, β], γ],

b. [α [$_{\gamma}$…β…]].

Option (a) will be selected if α is primitive, has an affix, and [α, β] satisfies LF-legibility (Section 4.10). In that case no further reconstruction operation will be applied to α. Otherwise option (b) is selected.

The operation travels downward on the right edge of α, targets primitive constituents β at the left that have an affix φ [$_{\beta}\varnothing$ φ] and drops the head/affix φ downstream if a suitable position is found. Dropping is implemented by fitting the head to the left of each node at the right edge. 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, has a local specifier, and can be selected in that position by a higher head. The left complex sister constitutes an acceptable specifier [XP, φ]. Heads are therefore 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 β that has EPP and is selected properly by α, i.e. [α β]. If β 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. The motivation is that the specifier could be filled in later by movement reconstruction. 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 [$_{\varphi}\varnothing$ γ] later when it travels downwards from the position it originally extracted φ, and applies the same algorithm to γ.

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.

4.7.3 Adjunct reconstruction (adjunct_reconstruction.py)

Adjunct reconstruction begins from the top of the phrase structure α and targets floater phrases at the left and to the right (e.g., DP at the bottom). If a floater is detected, it will be dropped. A floater has the following necessary properties:

(58) *A definition of a floater*

XP is a *floater* if and only if

(i) it is complex;

(ii) it has not been floated already;

(iii) it has a tail set;

(iv) floating is not prevented by feature [−FLOAT];

(v) external tail-head test fails.

Condition (iv) 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. This is another possibility that applies in some cases (genitive arguments) but fails in other (e.g. genitives inside deverbal nominals). Suppose α satisfies conditions (i-iv) above. Then if the tail-head features (Section 4.6.11) are not satisfied (Section 4.7.4), α must be subject to adjunct reconstruction and is designated as a floater.

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. In both cases, it is judged to be in a "wrong" position.

After a floater is detected, it will be dropped. Dropping is implemented by first locating the closest finite tense node. 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 involves three conditions:

(59) *Fitting a floater*

α can be fitted into position P if and only if

(i) tail-head features are satisfied (Section 4.7.4),

(ii) P is not the same position where α was found,

(iii) P is not a local specifier position.

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 once a suitable position is found. This will allow it to be treated correctly by selection, labeling and so on.

4.7.4 External tail-head test (*external_tail_head_test*)

External tail-head is defined in the following way.

(60) A head α's tail features are checked if either (a.) or (b.).

a. The tail-features are checked by a c-commanding head;

b. α is located inside a projection of a head having the tail features.

Tail features are checked in sets: if a c-commanding (a) or containing (b) 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). There are several reasons why checking must be nonlocal, long-distance structural case assignment and checking of negative polarity items being a few.

4.7.5 A´/A-reconstruction Move-1 (phrasal_movement.py)

Suppose A´/A-reconstruction (Move-1) 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 separate are checked, each leading into different action:

(61) *The three conditions of A/A-bar reconstruction*

(i) If the head α 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;

(ii) If the head α has the EPP property and has a specifier or several, they are stored into the memory buffer;

(iii) If the head α 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.

The phrase structure is explored, one head at a time, checking all three conditions for each head.

Option (i): fill in the SPEC position. If α does not have specifiers, a constituent is selected from the memory buffer if and only if either (1) α has a matching specifier selection feature (e.g., v selects for DP-specifier) or (2) α 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 (2) is not yet fully implemented, as the generalized EPP mechanism involved (Brattico 2016; Brattico, Chesi, and Suranyi 2019) is not formalized explicitly. An additional possibility is if an existing specifier of α 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 (ii): 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 α is an EPP head: thematic constituents are not targeted. Let us examine the single specifier case first.

A specifier refers to a complex left aunt constituent βP such that [βP [α_{EPP} XP]] and βP has not been moved already. If α has the generalized EPP feature, βP will undergo A-reconstruction (local successive-cyclicity, Section 4.7.6); otherwise it will be put into memory buffer. The latter option leads possibly into long-distance reconstruction (A´-reconstruction).

If βP has criterial features (which are scanned from it), formal copies of these features are stored to α. 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. This means that if F is the original criterial feature, then uF is a formal trigger of movement and iF is the semantically interpretable scope marker/operator feature. Lexical redundancy rules and parameters are applied to α to create a proper lexical item in the language being parsed (α might have language-specific properties). In addition, the label of α will be also copied to the new head, implementing the "inverse of feature inheritance." If α has a tail feature set, an adjunct will be created. 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 non-adjuncts, 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. 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. This operation is required when a relative pronoun creates a relative operator, transforming the resulting phrase into a relative clause adjunct.

Option (iii): fill in the complement position from memory. A complement for head α is merged to Comp,αP from the memory buffer if and only if (1) α is a primitive head, (ii) α does not have a complement or α has a complement that does not match with its complement selection, and (iii) α 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 4.7.7).

4.7.6 A-reconstruction (*A_reconstruct*)

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 the DP does not have any criterial features and α has the generalized EPP property: we assume that DP has been moved locally to satisfy this feature.

4.7.7 Extraposition as a last resort (extraposition.py)

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⟩]. 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 (Brattico 2012) 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].[5] 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 4.7.8). 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.

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:

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

This interpretation is wrong. 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 (63).

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

[5] The notion "is adjoinable" (*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.

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*'.

4.7.8 Adjunct promotion (adjunct_constructor.py)

Adjunct promotion 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." Decisions concerning what will be an adjunct and non-adjunct cannot be made in tandem with consuming words from the input. The operation is part of reconstruction. 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? Is it possible to promote a head to an adjunct while leaving its complement and specifier behind? It is obvious that the complement cannot be left behind. 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, and the model is currently unstable with respect to this issue, having undergone several revisions. The matter must be examined in detail later.

4.8 Agreement reconstruction Agree-1 (agreement_reconstruction.py)

Agreement reconstruction (Agree-1) is attempted after all movement has been reconstructed. The operation goes downstream from α and targets any primitive head to the left that has unvalued φ-set (φ_) and requires valuation (does not have feature -VAL). That triggers Agree-1. Such heads H attempt to *acquire* φ-features (*acquire_phi*). Acquisition of φ-features consists of two steps: (i) acquisition from φ-features from within the sister of H and (*acquire_from_sister*) and, if unvalued features remain, (ii) acquisition from the edge (*acquire_from_edge*). Operation (i) triggers phase-bounded (*is_phase*) downward search for left phrases with label D and collects all valued φ-features from the first such element, but with the exception of D-features that can only be acquired from the edge (Brattico 2019d). Phases are v, C, Force and BE. Operation (ii) targets the first phrase from the edge in a top-down order (adjunct or non-adjunct) with label D (*edge_for_Agree*) and obtains φ-features from it. Notice that the definition of 'edge' includes the pro-element, if present, at H itself. That element is only consulted if no phrasal specifier exists; the definition of generalized specifier is such that only if there is no phrase at SPEC will the pro-element be included.

Acquired φ-featured are then *valued* to the head. Denote a φ-feature of the type T with value V as [PHI:T:V]. An acquired φ-feature {PHI:T:v} can only be valued at H if (i) H contains {PHI:T:_} and (ii) no conflicting

feature {PHI:T:v´} exists at H. Condition (ii) leads into *φ-feature conflict* which, when present, crashes the derivation at LF. Thus, condition (ii) does not terminate the operation, but is illegitimate at LF.

4.9 Morphological and lexical processing (morphology.py)

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 literal 'space' has 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 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. 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), φ-features (e.g., PHI:NUM:SG) and others. The number and type of lexical features is not restricted by the model.

4.10 LF-legibility (LF.py)

4.10.1 Introduction

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. It constitutes an internal "perceptual mechanism" that ensures that what is being generated makes sense semantically.

4.10.2 LF-legibility tests

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).

4.10.3 LF-recovery (*LF-recovery*)

LF-recovery is triggered if an unvalued phi-feature occurs at the LF-interface. In the current implementation the operation (*LF-recovery*) returns a list of possible antecedents for the triggering head which are then provided in the results and log files. LF-recovery depends on the nature of unvalued phi-features, as follows.

(1) If both number and person features remain unvalued at H, this triggers 'standard control', which engages in an upstream path formation (upstream walk) and evaluates whether the sisters of nodes reached in this way evaluate as possible antecedents (*is_possible_antecedent*). The operation is blocked by v* head (head with 'sem:external'). A possible antecedent α must satisfy two conditions: (i) α cannot be a copy of an element that has been moved elsewhere and (ii) α must check all semantically relevant phi-features of H (*is_possible_antecedent*). Semantically relevant features are (by stipulation) number, person and definiteness. If no antecedent is found, generic interpretation is generated.

(2) If only D_ remains unvalued, then the above mechanism comes with three exceptional properties. (i) If search fails, the expression evaluates as ungrammatical; (ii) if there is a local specifier that is not a DP, generic interpretation is generated; (iii) v* does not block search.

(3) If only person remain unvalued, a radical pro-drop profile is activated which like (1) but results in discourse antecedent when LF-recovery fails.

References

Baker, Mark. 1996. *The Polysynthesis Parameter*. Oxford: Oxford University Press.

Brattico, Pauli. 2012. "Pied-Piping Domains and Adjunction Coincide in Finnish." *Nordic Journal of Linguistics* 35:71–89.

Brattico, Pauli. 2016. "Is Finnish Topic Prominent?" *Acta Linguistica Hungarica* 63:299–330.

Brattico, Pauli. 2018. *Word Order and Adjunction in Finnish*. Aarhus: Aguila & Celik.

Brattico, Pauli. 2019a. *A Computational Implementation of a Linear Phase Parser. The Framework and Technical Documentation*. Pavia.

Brattico, Pauli. 2019b. "Finnish Word Order: Does Language Comprehension Matter?" *Manuscript Submitted*.

Brattico, Pauli. 2019c. "Finnish Word Order and Morphosyntax." *Manuscript*.

Brattico, Pauli. 2019d. "Subjects, Topics and Definiteness in Finnish." *Studia Linguistica* 73(X):1–38.

Brattico, Pauli. 2020a. "Computational Linguistics as Natural Science and the Study of Romance Clitics." *Manuscript Submitted*.

Brattico, Pauli. 2020b. "Null Arguments and the Inverse Problem." *Submitted*.

Brattico, Pauli. 2020c. “Predicate Clefting and Long Head Movement in Finnish.” *Submitted*.

Brattico, Pauli and Cristiano Chesi. 2020. “A Top-down, Parser-Friendly Approach to Operator Movement and Pied-Piping.” *Lingua* 233:102760.

Brattico, Pauli, Cristiano Chesi, and Balasz Suranyi. 2019. “EPP, Agree and Secondary Wh-Movement.” *Manuscript*.

Chesi, Cristiano. 2004. “Phases and Cartography in Linguistic Computation: Toward a Cognitively Motivated Computational Model of Linguistic Competence.” Universita di Siena, Siena.

Chesi, Cristiano. 2012. *Competence and Computation: Toward a Processing Friendly Minimalist Grammar*. Padova: Unipress.

Chomsky, Noam. 1965. *Aspects of the Theory of Syntax*. Cambridge, MA.: MIT Press.

Chomsky, Noam. 1995. *The Minimalist Program*. Cambridge, MA.: MIT Press.

Chomsky, Noam. 2000. “Minimalist Inquiries: The Framework.” Pp. 89–156 in *Step by Step: Essays on Minimalist Syntax in Honor of Howard Lasnik*, edited by R. Martin, D. Michaels, and J. Uriagereka. Cambridge, MA.: MIT Press.

Chomsky, Noam. 2001. “Derivation by Phase.” Pp. 1–37 in *Ken Hale: A Life in Language*, edited by M. Kenstowicz. Cambridge, MA.: MIT Press.

Chomsky, Noam. 2005. “Three Factors in Language Design.” *Linguistic Inquiry* 36:1–22.

Chomsky, Noam. 2008. “On Phases.” Pp. 133–66 in *Foundational Issues in Linguistic Theory: Essays in Honor of Jean-Roger Vergnaud*, edited by C. Otero, R. Freidin, and M.-L. Zubizarreta. Cambridge, MA.: MIT Press.

Ernst, Thomas. 2001. *The Syntax of Adjuncts*. Cambridge: Cambridge University Press.

Holmberg, Anders, Urpo Nikanne, Irmeli Oraviita, Hannu Reime, and Trond Trosterud. 1993. “The Structure of INFL and the Finite Clause in Finnish.” Pp. 177–206 in *Case and other functional categories in Finnish syntax*, edited by A. Holmberg and U. Nikanne. Mouton de Gruyter.

Jelinek, Eloise. 1984. “Empty Categories, Case and Configurationality.” *Natural Language & Linguistic Theory* 2:39–76.

Phillips, Colin. 1996. “Order and Structure.” Cambridge, MA.

Phillips, Colin. 2003. “Linear Order and Constituency.” *Linguistic Inquiry* 34:37–90.

Salo, Pauli. 2003. “Causative and the Empty Lexicon: A Minimalist Perspective.” University of Helsinki.