Computational implementation of a linear phase parser. Framework and technical documentation

(version 6.9)

2019

(Revised January 2021)

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 linear phase parser-grammar. The parser-grammar assumes that the core computational operations of narrow syntax are applied incrementally 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).

1 Introduction

This document describes a computational Python based implementation of a linear phase parser that was originally developed and written by the author while working in an IUSS-funded research project between 2018-2020, in Pavia, Italy, and then continued as an independent project.[1] The algorithm is meant as a realistic description of the information processing steps involved in real-time language comprehension. This document describes properties of the version 6.9, which keeps within the framework of the original work but provides improvements, corrections and additions. The most significant change with respect to the previous version, most notably 6.x, was the inclusion of a more rigorous framework for addressing performance-related properties, which is now reflected also in the documentation. This change led to a number of other changes, for example, the overall introduction was removed, and its contents moved to the section dealing with performance.

This document does not substitute for a scientific treatment of the topics covered. There are very few references, and the topics are organized into a form of a tutorial that only illustrates the core issues by using simplest possible examples. Proper scientific discussion of these issues can be found from published sources.

---

2   The linear phase framework

2.1   The framework

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 the mapping from linguistic sensory objects into sets of possible meanings is called the parser, or perhaps more broadly as *language comprehension*. We assume that efficient and successful language comprehension must be possible from contextless and unannotated sensory input.

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, marginality or acceptability. 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. Indeed, processing of ungrammatical inputs have very little practical value. They cannot be ignored within a context of an empirical study, however.

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 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 (as in, e.g., *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. 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 way to what is observed

in reality. If, for example, the parser engages in astronomical garden-path derivations when no native speaker exhibits such inefficiencies, 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. Performance properties are discussed later in this document after we have examined the basic assumptions concerning grammar and grammatical representations.

Language production utilizes motoric programs that allow the speaker to orchestrate complex motoric sequences 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 responsible for the perception of the same units.

Language comprehension is viewed in this study as a cognitive information processing mechanism that processes information beginning from linguistic sensory input and ending up with a set of semantic interpretations. Sensory input is conceived as a linear string of phonological words that may or may not be associated with prosodic features. Lower-level processes, such as those regulating attention or separating linguistic stimuli from other information such as music, facial expressions or background noise, is not considered. Because the input consists of phonological words, it is presupposed that word boundaries have been correctly worked out during lower-level processing. Since the input is represented as a linear string, we will also take it for granted that all phonological words have been ordered linearly and that this ordering is well-defined and involves no overlap or any type of ambiguity.

The output contains a set of semantic interpretations. The output must constitute a set because the input can be ambiguous. Sentence *John saw the girl with a telescope* may mean that John saw a girl who was holding a telescope, or that John saw, by using the telescope that he himself had, the girl. This means that the original sensory input must be mapped to at least two semantic interpretations in this particular case. These semantic interpretations must, furthermore, provide interpretations that agree with native speaker intuition. In this way, we can say that the language comprehension model is descriptively adequate.

The ultimate nature of semantic representations is a controversial issue. It becomes even more challenging when working within a fully computational theory. Whatever assumptions one makes must be captured in computational form. One way around this problem, partially adopted here, is to focus on selected aspects of semantic interpretations and try to predict them on the basis of the input string. For example, we could decide to focus on the interpretation of thematic roles and require that the language comprehension model provides for each input sentence a list of outputs which determine which constituents appearing in the input sentence has which thematic roles. For example, we could require that the model provides that *John admires Mary* is processed so that John is judged the agent, Mary the patient. The advantage is that we can predict semantic intuitions in a selective way without taking a strong stance towards the ultimate nature and implementation of the semantic notions. Another advantage is that we can focus on selected semantic properties without trying to understand how the semantic system works as a whole.

A third advantage of this approach is that we do not need to decide a priori what type of linguistic structures will be used in making these semantic predictions. We can leave that matter open for each theory to settle and simply require that correct semantic intuitions *are* predicted. Of course, any given model must ultimately specify how these predictions are generated. I will adopt in this study the standard assumption in the present generative theory and assume that the input sentences are interpreted semantically on the basis of representations at a *syntax-semantics interface*. The syntax-semantic interface is considered a level of representation, perhaps ultimately a neuronal switchboard, at which all phonological, morphological and syntactic information processing has been completed and semantic processing begins. It is also called Logical Form or *LF-interface* in short. I will use both terms in this document. This means, then, that the language

comprehension model will map each input sentence into a set of LF-interface objects, which are interpreted semantically.

All these assumptions are summarized in Figure 1 which represents the overall framework we will use in implementing properties of human language comprehension.
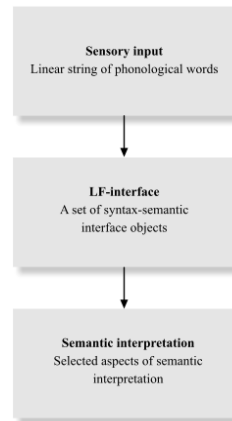
Figure 1. The general framework, in which a sensory input, conceived as a linear string of phonological words, is mapped into a set of LF-interface objects that generate semantic interpretations. The output is compared with the gold standard provided by a native speaker or several.

Notice the abstract nature of these assumptions: almost any model can be interpreted in these terms. Still, there are alternatives. If we assume that the input is something less than a linear string of phonological words, then the model must include additional lower-level information processing mechanisms that can preprocess such stimulus, perhaps ultimately the acoustic sound waves, into such a form that it can be used to activate lexical items in a specific order. We can also assume more, for example, by providing the model additional information concerning the input words, such as their morphological decompositions, morphosyntactic structure or part-of-speech (POS) annotations. To the extent that such information is provided, the model would then not explain how a native speaker generates such knowledge. Because Figure 1 does not allow such annotation, the model must consequently contain information processing steps which interpret the morphological structure, morphosyntactic features and part-of-speech information from bare phonological words.

It is not possible to construct a model of language comprehension without assuming that there occurs a point at which something that is processed from the sensory input is used to construct a semantic interpretation. A model that does not posit any such a point would be incomplete, processing sensory inputs without generating any meaning. A syntax-semantic interface seems to be a priori necessary. Different theories differ in where they position that interface, and what properties it has. It is possible to assume that the interface is located relatively "close to the surface" and operates with linguistic representations that have been generated from the sensory objects themselves by applying only a few operations. Meaning would then be read off from relatively shallow linguistic representations. An alternative is a theory in which the sensory input is subjected to much processing before anything reaches this stage. We can build the model by assuming that there is only one syntax-semantic interface, or several, or that the linguistic information flows into that system all at once, or in several independent or semi-independent packages. The only way to compare these alternatives, and many others that are surely imaginable, is to examine to what extent they will generate correct semantic interpretations for a set of input sentences; it is pointless to try to use any other justification or argument either in favor or against any specific model or assumption. In general, then, it makes no sense to try to posit any further restrictions or properties to the syntax-semantic interface apart from its existence.

Many linguists, including generative linguists, have treated the comprehension perspective as irrelevant, misleading or even empirically wrong. It is sometimes considered as being involved with (irrelevant, uninteresting) performance matters. Yet, this strange neglect only seems to concern a tiny minority of linguists competing in the construction of more and more exotic abstract grammars; everybody else seem to take a neutral and, in my view, also the correct stance on the matter, which is that by examining how the human brain processes language we can potentially learn something useful about how the brain and language works. If we can construct a observationally, descriptively and psycholinguistically adequate model for language comprehension then surely that cannot be an irrelevant undertaking.

2.2   Computational linguistics methodology

In this study it is assumed that a scientific hypothesis must be justified by deducing observed facts from the proposed hypothesis. The method, although routinely used in the hard sciences, is rarely if ever used in linguistics and therefore requires a brief comment.

The methodology starts by narrowing down a dataset based on the interests of the particular study, whether it be scattering patterns in a particle accelerator, celestial orbits or chemical reactions. In linguistics, a typical dataset contains grammatical and ungrammatical expressions paired with their meanings and other attributes, but it could involve actual use patterns, communicative intuitions, or pragmatic presuppositions. This dataset is then captured by developing a formal theory. Once the theory is set up, an attempt will be made to calculate its empirical consequences, which are then compared with the dataset. For example, if the calculated results of the differential equations do not match the observed locations of the celestial objects, then the theory is modified in some way, and the calculations are performed anew. The linear phase theory is a formal theory in this sense. It consists of a set of assumptions or axioms, expressed and formalized in a machine-readable language, and the logical consequences of the assumptions or axioms are compared against empirical reality by letting the computer to perform the required calculations. The theory predicts grammaticality judgments, semantic interpretations and performance properties for any given set of natural language sentences, in any language.

The difference in expression a linguistic theory in an ad hoc pseudo-formalism or in a machine-readable format is very small. Much of the content of the actual Python based formalization will be directly understandable for anybody working with formal grammars. Consider the following code snipped defining the notion of "head" for any binary branching phrase structure, as written in the current linear phase model. "Self" in the code refers to the constituent whose head we are computing, as it will be changed at run-time to whatever constituent we happen to be considering.

```
# Definition for head (also label) of a phrase structure
def head(self):
    if self.is_primitive():
        return self
    if self.left_const.is_primitive():
        return self.left_const
    if self.right_const.is_primitive():
        return self.right_const
    if self.right_const.externalized():
        return self.left_const.head()
    return self.right_const.head()
```

Expressed in English, this definition says the following: Suppose we take a phrase structure α (*self* in the above function). If α is primitive, then the label of α will be α; if α has two constituents and the left constituent is primitive, then that constituent will be the label; if not, and the right constituent is primitive, then the right

constituent will be the label. If the right constituent is complex but is not an adjunct, then we apply the rule recursively to it; otherwise we apply the rule recursively to the left constituent. If we compare this English based description with the code above, they look almost identical; in fact almost exact 'wording' can be found from the code. The fourth line in the code, for example, reads *if self.left_const.is_primitive*, which reads in plain English as "If the left constituent is primitive". Thus, there is nothing exotic or impenetrable in this methodology. The benefits, on the other hand, are plainly obvious: we can employ this formalization to compute the label of any constituent in an instant, and the computation is completely devoid of any errors or omissions. Not only that, but the theory is also unambiguous: every question has an answer, every constituent has an unambiguous head, every edge case can be calculated, no case need to slip out of our purview, every change or alternative model can be tested in an instant, and so on.

Why these advantages have not been harnessed is hard to say. In the 1950s it was commonplace to read authors advocating such methods, and for the exact right reasons already mentioned. An influential 1993 textbook on mathematical linguistics authored by Barbara H. Partee, Alice ter Meulen and Robert E. Wall correctly states that linguistics theories are made up of axioms whose theorems are the observed facts. On possible concerns is that the amount of computations implied in any nontrivial linguistic study is so huge that it makes no sense to perform them all with paper and pencil, yet this obstacle can be easily removed by using modern computers. We can put a fairly accurate quantitative number on these claims. It takes the current linear phase algorithm some 250000 linguistic operations (i.e., Merge, Move, Agree) to verify that a set of 1360 expressions conform with the model, which is clearly beyond any human calculator or feasible research project relying solely on human resources. For a standard laptop computer running suboptimal Python code, the operation takes little more than two minutes. Moreover, the output is always the same and therefore completely free of errors: the output is always exactly what the researcher has written into the axioms.

2.3   Merge-1

Linguistic input is received by the hearer in the form of sensory stimulus. We can first 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 sequence of abstract syntactic interpretations for the input string received

through the sensory systems in order to find 'who did what to whom'. These interpretations and the corresponding representations might be lexical, morphological, syntactic and semantic. One fundamental concern is to recover the *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 α * β, Merge yields [α, β](1).

(1)   John * sleeps.

     ↓   ↓

   [John, sleeps]

The first line represents the sensory input consisting of a linear string of phonological words, and the second line shows how these words are put together. What do we mean by saying that they are "put together"? The assumption in the linear phase theory is that while the two words in the sensory input are represented as two completely independent objects once they are put together in syntax in a manner shown in (1) they are represented as being part of the same linguistic chunk. Thus, at this point we can attend to both of them, manipulate them as part of the same representation, and in general perform operations that takes them both into account. All operations that occur before this point are performed on individual elements in isolation, as if they were the only thing that existed.

Example (1) suggests that the syntactic component combined the phonological words themselves. While this is a possibility, it is not linguistically useful. The laws and principles that cover these operations are not sensitive exclusively to phonological properties, such as the fact that *John* begins with /j/. For example, in order to process this chunk further it is important to know the lexical categories or labels of the elements (e.g., N, V) that appear in it. We must also know that *sleep* is an intransitive verb and does not accept a patient argument in any canonical sentence. There are further properties that are also important, though less obvious: in order to match the form *sleeps* with *John* we must know that the subject is in third person singular. Finally, at some point we have to get acquainted also with the meanings of these words. The linear phase theory assumes, therefore, that the operation illustrated in (1) is mediated by *lexicon*: a storage of linguistic

information that is activated on the basis of the original phonological words. At this point it is sufficient to think of this as an enrichment operation: the lexicon will enrich the phonological words in the input with further linguistic knowledge concerning the words, such as lexical categories (noun, verb), inflectional features ('third person singular') and meaning ('John', 'sleeping'). This enrichment process is implemented by assuming that the lexical items are sets of features, including the original phonological features. We should *not* think of the two elements as being the original phonological words.

The resulting complex chunk [α β] is assumed to be asymmetric: it has a *left constituent* and a *right constituent*. The terms "left" and "right" are mnemonic labels and do not refer to concrete leftness or rightness at the level of neuronal implementation. Their purpose is to distinguish the two constituents from each other. They are related to leftness indirectly: since we read the sensory input from left to right, (1) implies that the constituent that arrives first will be the left constituent. Thus, we could think of the left constituent as the "first constituent." It is important that this information is preserved inside syntax. It allows the language comprehension system to use original linear order in its computations. Of course, it is not a priori true that this asymmetric is recorded by means of asymmetric Merge. We could assume that Merge is symmetric, perhaps a set, and then encode the linear order by using other type of formal markings. The matter is hard to argue either way. In both systems the asymmetry must be coded into the representation at the moment the sensory input is chunked into complex representations, for otherwise the information will be lost permanently.

The assumption that Merge 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 time of 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 in the sensory input and thus derive the structure "backwards," as if were.

Notice that when the parser assembles (1) [*John*, *sleeps*], it does not yet know that the next word will be *furiously*. It could have been some other word, such as *and*, *because*, *much* or *every (day)*. These correspond to input sentences such as *John sleeps and sleeps forever, John sleeps because he was very tired, John sleeps much, John sleeps every day ten hours*. The human language comprehension system, because it works incrementally and thus one word at a time, cannot wait for the next word in order to create a syntactic interpretation for the words that it has already seen. If we assumed this to be the case, then it would have to read all words before attempting to interpret them syntactically and/or semantically, which is clearly not the case. We construct an interpretation for the sentence in real time, as it is presented, not after every word has been registered and stored somewhere as idle units waiting to get processing. Therefore, the system merges-1 the two first words into a tentative representation before the third and fourth words are considered. The ultimate syntactic interpretation of the whole sentence is generated as a sequence of *partial* phrase structure representations, first [*John*], then [*John*, *sleeps*], then [*John* [*sleeps*, *furiously*]], and so on, until all words have been consumed from the input. See also Section 3.3.1.

Several factors regulate Merge-1. One concern is that the operation creates a representation that is in principle ungrammatical and/or uninterpretable. Alternative (2)(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, assumed here, 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 indeed 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.
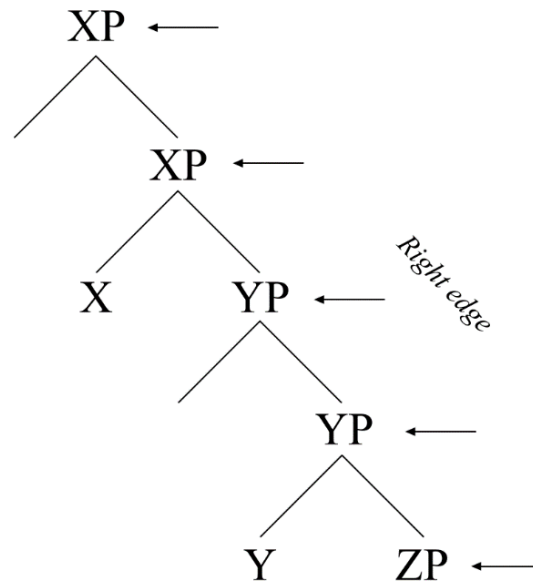
Figure 2. The right edge of a phrase structure. Nodes XP, YP and ZP are possible attachment sites. This condition follows from the assumption that linearization in language production is always top-down/left-right.

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 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 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 algorithm will 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 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.

The basic mechanism can be illustrated with the help of a standard garden path sentence such as (4).

(4)

a.   The horse raced past the barn.

b.   The horse raced past the barn fell.

Reading sentence (b) involves extra effort when compared to (a). Integrating the last word *fell* into the structure is exceptionally difficult. This is because at the point where the incremental parser encounters the last for *fell*, it has (under normal language use context) already created a partial representation for the input in which *raced* is interpreted as the past tense finite verb, hence the assumed structure is [$_{DP/S}$ *The horse*][$_V$ *raced*][$_{PP}$ *past the barn*] (DP/S = a subject argument, V = verb, PP = preposition phrase). Given this structure, there is no legitimate right edge position into which *fell* could be merged. Once all these solutions have been found impossible, the parser backtracks and considers if the situation could be improved by merging-1 *barn* into a different position, and so on, until it ultimately discovers a solution in which *raced* is interpreted as a noun-internal participle. The correct interpretation is [$_{DP/S}$ *The horse raced past the barn*][$_V$ *fell*]. The important thing to notice here is that this explanation presupposes that all solutions at each stage are ranked, so that they can be explored recursively in a well-defined order. Thus, at the stage at which raced is merged, the two solutions *raced* = finite verb and *raced* = participle are ranked so that the former is tried first.[2]

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

---

[2] This backtracking system is not assumed to be psycholinguistically realistic. It is used here as a baseline mechanism in order to be able to generate all possible legitimate syntactic interpretations for any sentence as determined by any given model of grammar. This makes it possible to evaluate if the competence model is correct.

must insert the constituent into a phrase structure and update the constituency relations accordingly (Section 5.1.1). 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.[3]

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 (5).

(5) 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, as shown in (6).

(6)   [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 (6), 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 a wrong verb form *sleep*. The result is a locally ungrammatical string **John sleep*. But because constituency relations can change in the

---

[3] 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 could 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.

derivation's future, we cannot rule this step out 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 2.5. It is important to keep in mind that in this 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 a core set of assumptions or at the very least that they are not mutual contradictory.

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. I will explore these issues later in Section 3.

2.4   The lexicon and lexical features

Consider next a transitive clause such as *John admires Mary* and how it might be derived under the framework as described so far (7).

(7) John   admires Mary

    ↓   ↓   ↓

    [John   [admires Mary]]
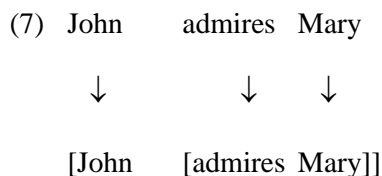
There is linguistic evidence that this derivation matches with the correct hierarchical relations between the three words (ignoring the details). The verb and the direct object form a constituent that is merged with the subject. We can imagine that this hierarchical configuration is interpretable at 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 is the opposite: now Mary is the one who admires John. But the fact that the verb *admire*, unlike *sleep*, can take a DP argument as its complement must be determined somewhere. This information cannot be read from the phonological word *admire*; in fact, the phonological shape of any word is close to being completely arbitrary.

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/or 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 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 (8), the test is passed: the label of the selecting item matches with the label of the new arrival.

(8) John   admires   Mary

    ↓   ↓   ↓

    [John   [admires   Mary]]

                COMP:D   D   *Mary* satisfies an "expectation".
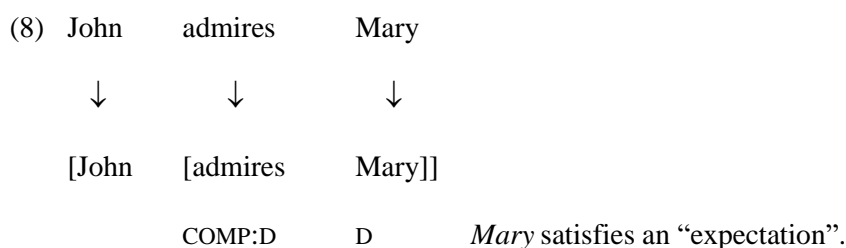
Let us assume the following conventions. Feature COMP:L means that the lexical item *licenses* a complement with label L, and !COMP:L says that it *requires* a complement of the type L. Correspondingly, −COMP:L says that the lexical item does *not* allow for a complement with label L.

There is a certain ambiguity in how these features are used. When the parser 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 2.3). But when the phrase structure has been completed and there is no longer any input to be consumed, the same features can now 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 or negative −COMP). 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. For present purposes, the crucial point is that using lexical features to guide language comprehension and to use them to evaluate grammaticality are two different things.

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 constructing an example. There are three options in (8): (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. We are left with two options (9)a-b.

(9)

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 left 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 all else being equal.

Notice that what constitutes a specifier selection feature will guide the selection of a possible left sister during comprehension. Because constituency relations may change later, we do not know which elements will constitute the actual 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 (10).

(10) 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 stand in the following configuration:

(11) 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 most configurations established during first pass parsing are subject the change later in the parsing derivation.

2.5   Phases and left branches

Let us consider the derivation of a slightly more complex clause (12).

(12)   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 for by (Brattico and Chesi 2020). We can formulate the condition tentatively as (13), but a more rigorous formulation will be given as we proceed.

(13) *Left Branch Phase Condition (LBPC)*

Derive each left branch independently.

All left branches are effectively thrown away from the working space once have been assembled and fully processed. If no future operation is able to affect a left branch, all grammatical operations (e.g. movement reconstruction) that must 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 still 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 "unrepairable" left branch can be either filtered out as unusable or at the very least be ranked lower.

2.6   Labeling

Suppose we reverse the arguments in (12) and derive (14).

(14) Mary   admires John's   mother

    ↓   ↓   ↓   ↓

    Mary   [admires [John's   mother]]

             COMP:D
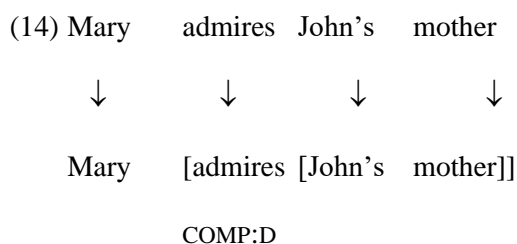
In this configuration the verb selects for a DP, but the complement selection feature refers to the label D of the complement. What is relationship between the label D and the phrase *John's mother* that occurs in the complement position of the verb in the above example? That relationship is defined 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"[4]).

The effects of the labeling algorithm can be perhaps best illustrated with the help of the artificial examples created from the linear string *a * b * c * d*. When fed with a string of empty lexical elements like these, the model produces the following syntactic interpretations (16).

---

[4] 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.

(16)

```
        aP                      aP                           dP
       /  \                    /  \                         /  \
      a    bP                 a    dP                      aP    d
          /  \                    /  \                    /  \
         b    cP                 bP    d                 a    bP
             /  \               /  \                         /  \
            c    d             b    c                       b    c
```

```
              cP                             dP
             /  \                           /  \
           aP    cP                       cP    d
          /  \  /  \                     /  \
         a   b c    d                  aP    c
                                      /  \
                                     a    b
```
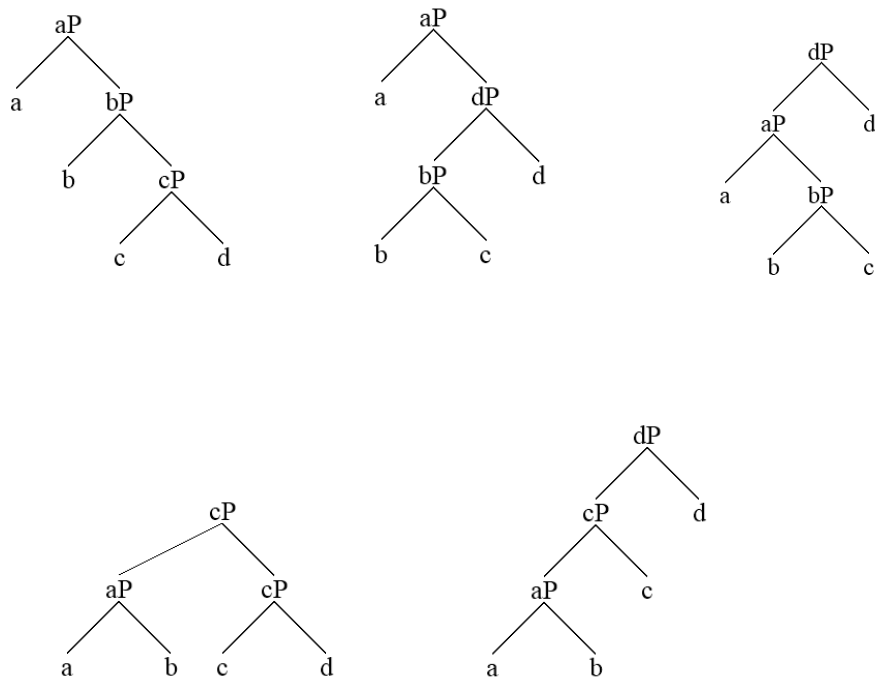
It is easy to see how the labeling algorithm works; it is easy to verify that they follow the algorithm just given. But consider again the derivation of (1), repeated here as (17).

(17) 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: (17) 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 (17) and other similar examples as exceptions. There is nothing exceptional or anomalous in (17). 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 surface lexicon, where proper names are decomposed into D + N structure. The structure of (17) is therefore (18).

(18)   John + sleeps.

       ↓   ↓

      D   N   sleeps

      ↓ ↓ ↓

    [$_{VP}$[$_{DP}$D N] sleeps]
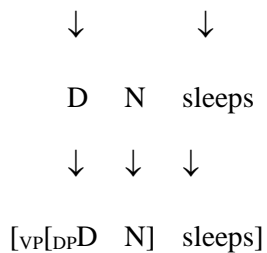
It is important to note that labeling presupposes that primitive elements, when they occur in prioritized (i.e., left) positions, always constitute heads. A head at the right constitutes a head if and only if there is a *phrase* at left. This happens in (18), which means that the whole phrase will come out as a verb phrase. The outcome will be the same if the verb is transitive, in which the structure and labels are provided in (19). A slightly more realistic output, provided by the parser, is shown in (20).

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

(20)
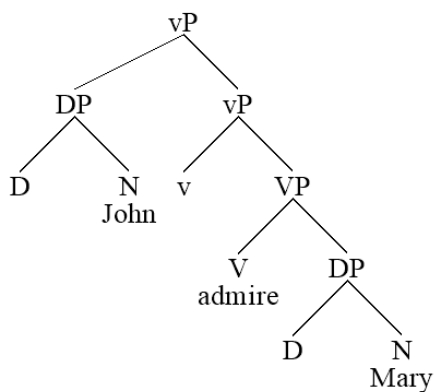
From the fact that labelling is determined by an algorithm over the phrase structure and that the structure is generated incrementally it follows that the label of any part of the partial representation may change as more words are integrated into the structure. The initial representation [*John*, *sleeps*] will be interpreted as a DP, but as soon as the pronoun is opened up into [D N] structure, the label changes into VP, as shown in (21).

(21)

a. [John, sleeps] = [$_{DP}$ D V] = DP (left constituent is primitive, hence the label)

b. [[D N] sleeps] = [$_{VP}$ DP V] = VP (left constituent is complex, hence the right constituent is the label)

This will affect the way these constituents are selected. The first representation can be selected as a DP, whereas the second as a VP. Once a complete phrase structure has been built, labels are locked and can no longer undergone any change.

2.7   Adjunct attachment

Adjuncts, such as adverbials and adjunct PPs, present a challenge for any incremental parser. Consider the data in (22).

(22) (Finnish)

a. *Ilmeisesti*   Pekka   ihailee   Merjaa.

    apparently   Pekka   admires Merja

    'Probably 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 and present a challenge for the parser

because the position of the adverb is completely unpredictable. 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. This hypothesis is illustrated in Figure 24.
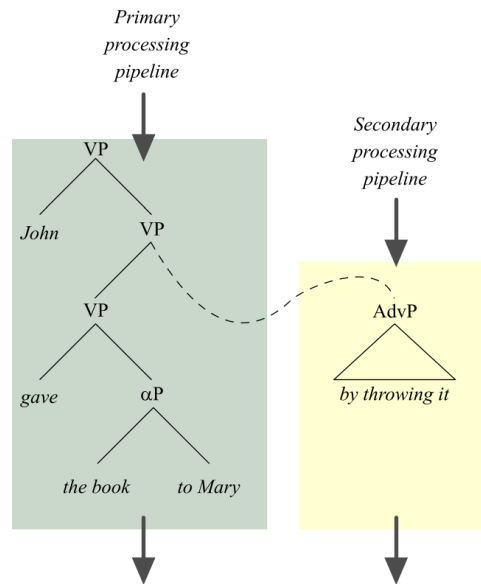
Figure 24. A nontechnical illustration of the way the linear phase comprehension algorithm processes adjuncts. Adjuncts are geometrical constituents that are "pulled out" of the primary working memory and are processed inside separate processing pipeline.

Thus, the labeling algorithm as specified in Section 2.6 ignores adjuncts. The label of (23) becomes V and not Adv: 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.

(23) [$_{VP}$ John [$_{VP}$[$_{VP}$ admires Mary] $\langle_{AdvP}$ furiously$\rangle$]]

Consider (24) next.

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

The adverb constitutes the sister of the verbal head V 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, then, the structure of (24) is [$_{VP}$[$_{DP}$ *John*] *sleeps*].

The fact that adjuncts are optional follows from 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, however, 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 a *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 (25). This analysis owes much to (Ernst 2001).

(25) Condition on tail-head dependencies

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

    a. H occurs inside a projection whose head K has F, or is K's sister;

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

Condition (a) is relatively uncontroversial. It states that a VP-adjunct must occur inside VP, or more generally, αP adjunct has to be inside a projection from α, where α is some feature. It does not restrict the position at which an adjunct must occur inside αP, only that it must occur inside αP. Therefore, both right-adjoined and left-adjoined phrases are accepted. Condition (b) allows some adjuncts, such as preposition phrases, remain in a low right-adjoined or extraposed positions in a canonical structure. If condition (b) is removed, all adjuncts will be reconstructed into positions in which they are inside the corresponding projections (reconstruction will be discussed later) or occur as their sister. The matter is controversial. Different definitions derive the fact differently. 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 2.9.4.

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 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 linked with a tail feature to the primary structure that provides considerable freedom, as provided in (25). 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, linking relations are typically 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. The process in which some phrase is promoted into an adjunct status is called *externalization*. The term refers to the fact that it is pulled or literally externalized out of the current syntactic working space.

2.8   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. Instead of providing a long list of specifier features, we capture this situation by an unselective specifier feature -SPEC:*, SPEC:* and !SPEC:*. The asterisk denotes the fact that the feature or label does not matter. Because the feature is unselective, it is not interpreted thematically, it cannot designate a canonical position, and hence the existence of this feature on a head triggers A´/A movement reconstruction (Section 2.9.1). This constitutes a sufficient (but not necessary) feature for reconstruction; see 2.9.1. 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.

2.9   Move-1

2.9.1   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 a

legible 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 properties of the verb. Example (26) 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.

(26) John   admires Mary

    ↓   ↓   ↓

   [John   [admires Mary]]

Example (27) shows a variation in which this is no longer the case.

(27) 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):

(28) [$_{VP}$ who   [$_{VP}$ Pekka   admires]]

The clause violates selection: there are two specifiers DP_1 and DP_2 at the left edge, one of which is not selected in any way, 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 from its canonical complement position, and this is the reason it also triggers complement selection failure. The parser must reverse-engineer this dislocation in order to create a representation that can be interpreted at the LF-interface and which satisfies all selection requirements of all lexical elements. These operations, which is called *reconstruction*, take place during *transfer*.

There is a strong temptation to assume, perhaps on the basis of the appearance of the interrogative word *who*, that this must be an interrogative clause, hence that (28) would be a wrong or at the very least an insufficient structure for this input string. We must resist this temptation: at the point at which the incremental parser is

consuming words from the input, it cannot know what the structure of the forthcoming sentence will be, not even when it encounters an interrogative pronoun. Interpretation (28) is the partial phrase structure when the comprehension system reads words *who * Pekka * admires* from the input and then realizes that there are no more words. Therefore, although it is clear that (28) is not an adequate representation for this input sentence, we cannot take this as a given; rather, we need to work out the exact computational principles that transform incomplete sentences of this type into something that is linguistically more plausible and also semantically interpretable. Furthermore, these operations must generate an interpretation for this sentence and others like it which correspond to the semantic intuitions obtained from native speakers. (28) is the phrase structure representation from which these computations and information processing steps must begin, since it contains everything there is in the input sentence.

Another common tactic is to deny that there is any need for reconstruction. We could take (28) and feed it directly, as it is, to the LF-interface component for semantic interpretation. It is, indeed, possible to hypothesize that these sentences are processed in this shallow form without reconstruction. Yet although possible in theory, this tactic does not solve the problem we face here, it only postpones it. The problem is that sentence (28) does not satisfy the complement selection features of the verb *admire*, which requires that it be complemented with a DP argument representing the patient, the target of admiration. We are assuming that sentences like ??*John admires* or *John admires to leave* are ungrammatical or at the very least less grammatical than *John admires Mary*. So how is this requirement satisfied by (28)? It is intuitively satisfied by the fact that the patient of admiring – the direct object of the verb – occurs in a noncanonical position at the beginning of the clause. There must therefore occur a process which relates the first element of the clause with the empty position at the end of the clause – and this is the reconstruction operation we are now addressing. Similarly, the main verb is preceded by two argument DPs in this clause, yet we cannot allow this to happen more generally. A sentence such as *John Mary admires* is again ungrammatical. The verb *admire* does not have two subjects or agents. The same reconstruction is needed to solve this puzzle.

Before addressing the specific formal mechanisms, it makes sense to ask a more fundamental question: why do many languages dislocate words and phrases? What is the interrogative pronoun doing at the first position of the clause?

The first-pass parse generated directly from any 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 the *spellout structure*. The spellout structure therefore creates 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, on the other hand, are often assumed to be near-universal. It is possible to translate English sentences into Finnish with little alteration in meaning, 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 in which most language-specific properties have been eliminated. This "normalization" is what the transfer achieves when it maps spellout structures into LF-interface representations. If this is true, then dislocation exhibited by sentences like (28) – 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.

We have seen that a standard interrogative in Finnish creates a spellout structure in which the sentence begins with two DPs, the interrogative pronoun and the subject element. Let us address this double specifier problem first. The linear phase algorithm solves it by generating a head between them (29).

(29) [Ketä   [C(*wh*)   [Pekka   ihailee   __]]]?

     wh   C(*wh*)

     who.par   Pekka   admires

     'Who does Pekka admire?'

The rationale is that the very reason the interrogative phrase was moved to the first position was that the interrogative 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. The moved element signals the "beginning" of the interrogative act. The mechanism has two steps. First, we must recognize that a head is missing. That is inferred from the existence of the two specifiers that form an illegitimate configuration at the LF-interface. The next step is to generate the label for the new head. This information is obtained from the *criterial feature*, i.e. from the fact that the element is an interrogative pronoun. This allows the comprehension system to infer both the existence and 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 itself must then be reconstructed back to its canonical LF-position to satisfy the complement selection for the verb *admire*.

(30) Who   does   John   admire   __?

       ——— Reconstruction ——————→

Reconstruction (called Move-1) works by copying the element occurring in the thematically wrong position and by 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 all 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 and only during transfer, thus when the spellout structure is send out to the syntax-semantics interface for interpretation. Transfer 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. This is necessary due to the left branch phase condition, discussed earlier. The third condition is if α is an adjunct. The two first conditions are expressed by (31)(Brattico and Chesi 2020), while the third was added later.

(31) Transfer α if and only if

      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. Specifically, 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.

2.9.2   A-reconstruction and EPP

A 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 next available specifier position downstream. This happens, for example, when the grammatical subject is reconstructed from SpecTP to SpecvP in an English finite clause, as shown in (32).

(32)

```
                        TP
              ┌──────────┴──────────┐
            DP_i                    TP
          ┌──┴──┐              ┌─────┴─────┐
          D     N             T            vP
               John                 ┌───────┴───────┐
                                  DP_i              vP
                                ┌──┴──┐         ┌────┴────┐
                                D     N         v         VP
                                     John              ┌───┴───┐
                                                       V       DP
                                                     admire  ┌──┴──┐
                                                             D     N
                                                                  Mary
```
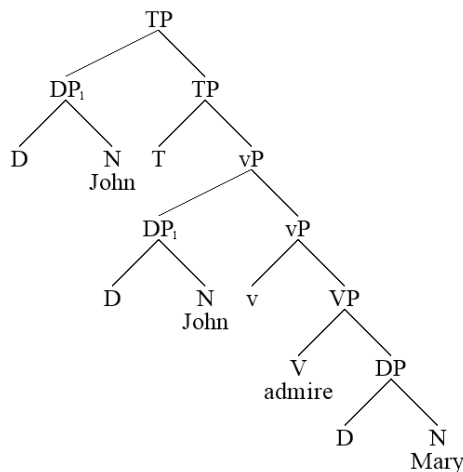
2.9.3   Head reconstruction

Many heads occur in noncanonical positions in the input string. Consider (33).

(33) 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 language comprehension algorithm 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 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 (34). Symbol "#" indicates that there is no word boundary between the morphemes/features.

(34) C(-kO) + #T_{inf} +   #V +   Pekka + ajatteli + että +   hänen + pitää

      C T_{inf} V   Pekka   thought   that   he   must

39

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.[5] 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} (35).

(35) $[_C \varnothing\ [_{T/inf} \varnothing\ V]]$ Pekka   ajatteli   että hänen   pitää   __.

                        Pekka   thought   that he   must

Representation (35) 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}.

(36) $[_C \varnothing\ [_{T/inf} \varnothing\ V]]$ Pekka   ajatteli   että hänen   [pitää $[[_{T/inf} \varnothing\ V]$ V]].

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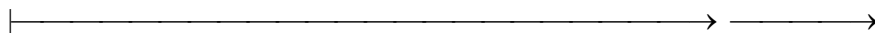
The operation fixes problems with cluttered heads in the input. It resembles phrasal movement reconstruction: it considers the cluttered components of the complex head to be in a wrong position and attempts to drop them into first positions available in which they could create a legitimate LF-object.

2.9.4   Adjunct reconstruction

Consider the pair of expressions in (37) and their canonical derivations.

---

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

(37)

a.

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

[Pekka [asked [we [to.admire Merja]]]]

'Pekka asked us to admire Merja.'

b.

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

[Merja [asked [we [to.admire Pekka]]]

'Pekka asked us to admire Merja.'

Derivation (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 SpecVP. 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, 2019b, 2020c, 2020a). See (Baker 1996; Chomsky 1995: 4.7.3; Jelinek 1984) for similar hypothesis. 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 (38).

(38) 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, viewed as 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

41

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 an adjunct and reconstruction into a position in which (38) is satisfied. In this way, the inversed subject and object can find their ways to the canonical LF-positions (39). Notice that because the grammatical subject *Pekka* is promoted into adjunct, it no longer constitutes the complement; the partitive-marked direct object does.

(39) [⟨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. We can see the effects of these assumptions by feeding the parser with a noncanonical Finnish OVS sentence (40). The result is (41).

(40) Merja-a   ihailee   Pekka.

   Merja-par   admire-3sg   Pekka.nom

   'When it comes to Merja, Pekka admires him.'

(41)

```
                        TP
                      ╱╱  ╲
                   DP₁      TP
                  ╱   ╲    ╱  ╲
                 D     N  T    vP
                     Merja   ╱╱ ╲
                          DP₂    vP
                         ╱  ╲   ╱  ╲
                        D    N v    VP
                          Pekka   ╱╱ ╲
                               DP₁    VP
                              ╱   ╲  ╱╲
                             D     N V  DP₂
                                Merja ihaile-  ╱ ╲
                                     'admire' D   N
                                                 Pekka
```
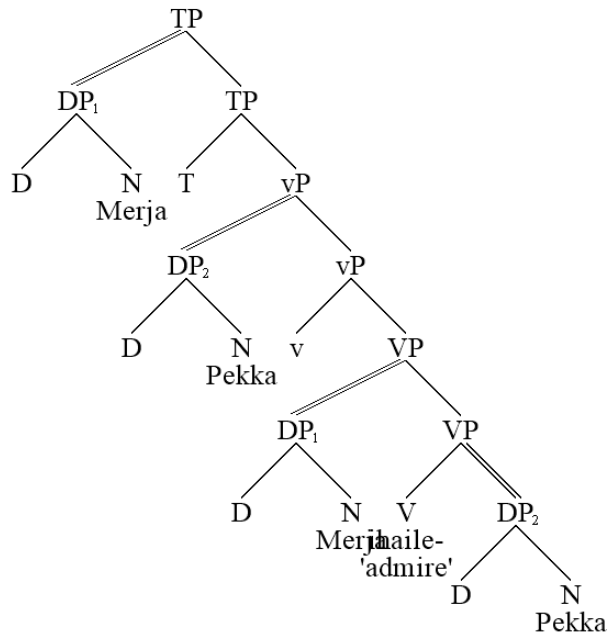
Adjunction is marked by double lines in the image output. Because the lower DP is reconstructed as an adjunct, V takes DP₁ as its sister and thus complement; DP₂ is invisible at the lower position.

The case system elucidated above is a simplification. To handle all relevant cases, a slightly more complex mechanism is required. However, the basic idea has remained the same in all subsequent iterations: case suffixes are reflections of tail-features that relate arguments to functional elements.

2.9.5   Ordering of operations

Movement reconstruction is part of transfer. Both A´/A-reconstruction and adjunct reconstruction presuppose head reconstruction; heads and their lexical features 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 (42).

(42) 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.

2.10 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. The surface lexicon (dictionary) first matches phonological words with morphological decompositions. A morphological decomposition consists of a linear string of morphemes $m_1 \# ... \# m_n$ that are separated and inserted into the linear input stream individually (43). Notice the reversed order.

(43) Nukku-u-ko   +   Pekka   →   Q   +   T/fin   +   V   +   Pekka

    sleep-T/fin-Q   Pekka   ↓   ↓   ↓   ↓

    sleep#T/fin#Q Merge^{-1} Merge^{-1} Merge^{-1} Merge^{-1}

    'Does Pekka sleep?'
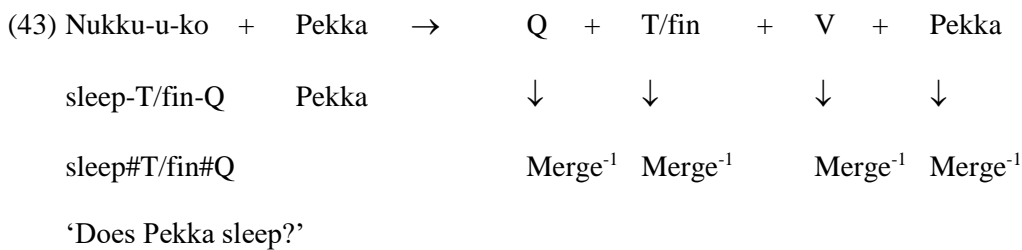
The lexical entry for the complex phonological word *nukkuako* is therefore 'sleep#T/fin#Q', where each morpheme *sleep-*, T/fin and Q are matched with lexical items (LIs) in the same lexicon. The lexicon has a hierarchical structure: one lexical entry can contain pointers to more primitive entries, which are linked with linguistic atoms or terminal elements, sets of features that have no further morphological decomposition. 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 and then inserted inside adjacent lexical items. That is, inflectional suffixes are representations of lexical features, whereas morphemes are representations of lexical items consisting of features.

Lexical features emerge from three distinct sources. One source is the language-specific lexicon, which stores information specific to lexical items 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.

2.11   Argument structure

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. It is read off from the properties of LF-interface objects as follows.

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 itself does not see the interpretation, it only sees the selection feature; the interpretation is read off from a representation generated by the model. 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 'at', 'to' or 'for') must be linked with verbs which describe 'directional' events (44). 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 can tail.

(44)

a. *Pekka   näki   Merjalle.

   Pekka   saw   to.Merja

   '#Pekka saw at Merja.'

b. Pekka   huusi   Merjalle.

   Pekka   yelled   at.Merja

   'Pekka yelled at Merja.'

The syntactic representation of an argument structure raises nontrivial 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*.

2.12 Agree-1

Most languages exhibit an agreement phenomenon illustrated in the example (45).

(45) John   [admire   +   *s*   Mary]

   3sg   3sg

Here the form of the verb depends in some manner on the properties of the subject proper name. Specifically, the third person features of the subject are reflected in the third person agreement marker on the finite verb. The term "agreement" or "phi-agreement" refer to a phenomenon in which the gender, number or person features (collectively called phi-features or φ-features in this document) of one element, typically a DP, covary with the same features on another element, typically a verb, as in (45). 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.

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 agreement, when it is possible in one of the three forms, is sensitive to structural conditions and constraints. 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. 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.

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 DP. 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 plays 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; not a plurality of things, for example. These features will be abbreviated as φ, but when we want to be more explicit we can 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 or covary with 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 missing and will be provide by the argument with which the predicate is linked with. This is shown in (46).

(46) Mary   [admires   John]

    D   N   T/fin

    ↓   ↓

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 (47). Agree-1 is applied only to heads with +VAL.

(47) 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 (in the lack of better term) to 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, if we wany to search for any, is to saturate the predicate by searching for an argument *from the sensory input*.

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 literally: in many languages with sufficiently rich agreement no overt phrasal argument is required. Example (48) comes from Italian.

(48) adoro   Luisa.

admire.1sg   Luisa

'I admire Luisa.'

Inflectional phi-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 as shown in (49).

(49) 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 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 inside a head as if it were a 'truncated pronominal element', call it pro. Sentence (50) should therefore be interpreted literally as 'I.admire Luisa'.

(50) adoro   Luisa.

admire.1sg   Luisa

admire.pro   Luisa

'I admire Luisa.'

Agree-1 is limited to local domain. 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 $\varphi\_$. 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. 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.

2.13 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) LF-recovery 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 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 =. The result is an interpretation in which John is both the agent of wanting and the agent of leaving:

(51) John wants to leave$_{\varphi\_ = John}$

    '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' (52).

(52) 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 is due Finnish third person agreement suffix being unable to value D_. Assume so. Then D_ triggers LF-recovery at LF, as shown by (53).

(53) Pekka   sanoi   että   nukkuu   hyvin.

      Pekka   said   that   sleep.3sg$_{D\_=\text{Pekka}}$ well

      'Pekka said that he (=Pekka) sleeps well.'

This illustrates a situation in which Agree-1 takes place but fails or perhaps succeeds only partially.

3   Performance

3.1   Human and engineering parsers

The linear phase theory is interpreted as a realistic model of the human language comprehension. Its behavior and internal operation should not be inconsistent with what is known concerning human behavior from psycholinguistic and neurolinguistic studies and, when it is, such inconsistencies must be regarded as defects in the model that should not be ignored rather than being judged as irrelevant for linguistic theorizing. In this section I will examine the neurocognitive principles behind the model, their implementation, and also examine them in the light of some experimental data. Proper scientific discussion of these topics can be found from the published literature.

3.2   Mapping between the algorithm and brain

Figure 32 maps the components of the model into their approximate locations in the brain on the basis of neuroimaging and neurolinguistic data.
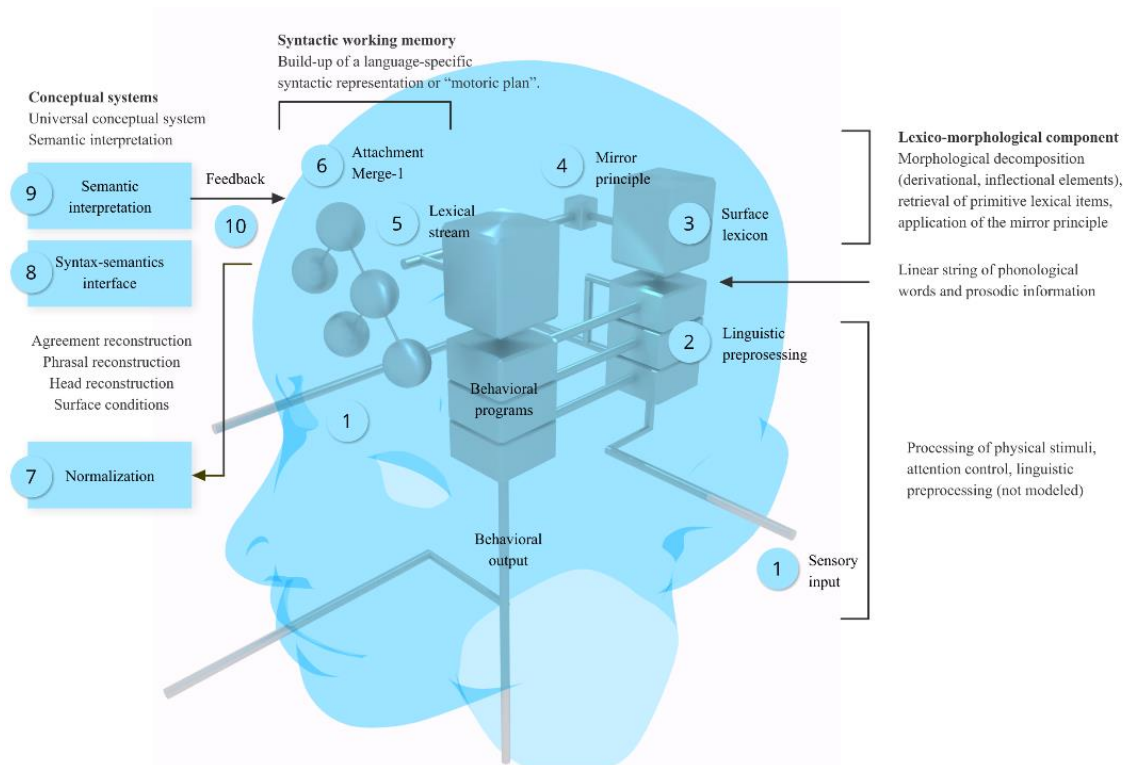
Figure 32. Components of the model and their approximate locations in the human brain. See the main text for explanation.

Sensory stimulus (1) is processed through multiple layers of lower-level systems responsible for attention control and modality specific filtering, in which the linguistic stimuli is separated from other modalities and background noise, localized into a source, and ultimately presented as a linear string of phonological words (2). The current model assumes (2) as its input. Brain imaging suggests that the processing of auditory linguistic material takes place in and around the superior temporal gyrus (STG), with further processing activating a posterior gradient towards the Wernicke's area that seems responsible for activating lexical items (3)(Section 2.4). Preprocessing is done in lower-level sensory systems that rely on the various modules within the brain stem. It is assumed that activated lexical items are streamed into syntax (5) after the application of the mirror principle (4). Construction of the first syntactic representation for the incoming stimulus is assumed to take place in the more anterior parts of the dominant hemisphere, possibly in and around Broca's region and the anterior sections of STG (6)(Section 2.3). It is possible that these same regions implement transfer as well (Section 2.9), as damage to Broca's region seem to affect transformational aspects of language comprehension. The syntax-semantic interface (LF-interface) is therefore quite conceivably also implemented within the

55

anterior regions and can be assumed to represent the endpoint of linguistic processing. There is very little neurolinguistic data on what happens after that point.

3.3   Cognitive parsing principles

3.3.1   Incrementality

The linear phase algorithm is an *incremental* parser, meaning that each incoming word is attached to the existing partial phrase structure as soon as it is encountered in the input. Each word is encountered by the parser as part of a well-defined linear sequence. No word is put into a temporal working memory to be attached at some later point, and no word is examined before other words occurring before it in the linearly organized sensory input. Apart from certain rearrangement performed by transfer normalization, no element that has been attached to the partial phrase structure can be extracted from it later.

There is robust evidence that the human language comprehension system is incremental. It is possible to trick the system into making wrong decisions on the basis of incomplete local information, showing that the parser does not wait for the appearance of further words before making parsing decisions. This can be seen from (54), in which the parser interprets the word *raced* as a finite verb despite the fact that the last word of the sentence cannot be integrated into the resulting structure.

(54)

a.   The horse raced past the barn.

b.   The horse raced past the barn fell.

The linear phase algorithm behaves in the same way: it interprets *raced* locally as a finite verb and then ends up with a dead end when processing the last word *fell*, backtracks, and consumes additional cognitive resources before it find the correct analysis. The following shows how the algorithm derives (b). Step (11) contains the first dead end; notice how the algorithm then immediately backtracks.

```
1    the
     the + horse
2    [the horse]
     [the horse] + T
3    [[the horse] T]
     [[the horse] T] + race
4    [[the horse] T(V)]
     [[the horse] T(V)] + past
5    [[the horse] [T(V) past]]
     [[the horse] [T(V) past]] + the
```

```
6    [[the horse] [T(V) [past the]]]
     [[the horse] [T(V) [past the]]] + barn
7    [[the horse] [T(V) [past [the barn]]]]
     [[the horse] [T(V) [past [the barn]]]] + T
8    [[the horse] [T(V) [past [[the barn] T]]]]
     [[the horse] [T(V) [past [[the barn] T]]]] + fell
9    [[the horse] [T(V) [past [the [barn T]]]]]
     [[the horse] [T(V) [past [the [barn T]]]]] + fell
10   [[the horse] [T(V) [[past [the barn]] T]]]
     [[the horse] [T(V) [[past [the barn]] T]]] + fell
11   [[[the horse]:11 [T [__:11 [race [past [the barn]]]]]] T]
     [[[the horse]:11 [T [__:11 [race [past [the barn]]]]]] T] + fell
12   [[[the horse]:12 [T [__:12 race]]] past]
     [[[the horse]:12 [T [__:12 race]]] past] + the
13   [[[the horse]:12 [T [__:12 race]]] [past the]]
     [[[the horse]:12 [T [__:12 race]]] [past the]] + barn
14   [[[the horse]:12 [T [__:12 race]]] [past [the barn]]]
     [[[the horse]:12 [T [__:12 race]]] [past [the barn]]] + T
15   [[[the horse]:12 [T [__:12 race]]] [past [[the barn] T]]]
     [[[the horse]:12 [T [__:12 race]]] [past [[the barn] T]]] + fell
16   [[[the horse]:12 [T [__:12 race]]] [past [the [barn T]]]]
     [[[the horse]:12 [T [__:12 race]]] [past [the [barn T]]]] + fell
17   [[[the horse]:12 [T [__:12 race]]] [[past [the barn]] T]]
     [[[the horse]:12 [T [__:12 race]]] [[past [the barn]] T]] + fell
18   [[[[the horse]:12 [T [__:12 race]]] <past [the barn]>] T]
     [[[[the horse]:12 [T [__:12 race]]] <past [the barn]>] T] + fell
19   [the [horse T]]
     [the [horse T]] + race
20   [the [horse T(V)]]
     [the [horse T(V)]] + past
21   [the [horse [T(V) past]]]
     [the [horse [T(V) past]]] + the
22   [the [horse [T(V) [past the]]]]
     [the [horse [T(V) [past the]]]] + barn
23   [the [horse [T(V) [past [the barn]]]]]
     [the [horse [T(V) [past [the barn]]]]] + T
24   [the [horse [T(V) [past [[the barn] T]]]]]
     [the [horse [T(V) [past [[the barn] T]]]]] + fell
25   [the [horse [T(V) [past [the [barn T]]]]]]
     [the [horse [T(V) [past [the [barn T]]]]]] + fell
26   [the [horse [T(V) [[past [the barn]] T]]]]
     [the [horse [T(V) [[past [the barn]] T]]]] + fell
27   [[the [horse [T [race [past [the barn]]]]]] T]
     [[the [horse [T [race [past [the barn]]]]]] T] + fell
28   [[the [horse [T [race [past the]]]]] barn]
     [[the [horse [T [race [past the]]]]] barn] + T
29   [[the [horse [T [race [past the]]]]] [barn T]]
     [[the [horse [T [race [past the]]]]] [barn T]] + fell
30   [[the [horse [T [race past]]]] the]
     [[the [horse [T [race past]]]] the] + barn
31   [[the [horse [T [race past]]]] [the barn]]
     [[the [horse [T [race past]]]] [the barn]] + T
32   [[the [horse [T [race past]]]] [[the barn] T]]
     [[the [horse [T [race past]]]] [[the barn] T]] + fell
33   [[the [horse [T [race past]]]] [the [barn T]]]
     [[the [horse [T [race past]]]] [the [barn T]]] + fell
34   [[the [horse [T race]]] past]
     [[the [horse [T race]]] past] + the
35   [[the [horse [T race]]] [past the]]
     [[the [horse [T race]]] [past the]] + barn
36   [[the [horse [T race]]] [past [the barn]]]
     [[the [horse [T race]]] [past [the barn]]] + T
37   [[the [horse [T race]]] [past [[the barn] T]]]
     [[the [horse [T race]]] [past [[the barn] T]]] + fell
38   [[the [horse [T race]]] [past [the [barn T]]]]
     [[the [horse [T race]]] [past [the [barn T]]]] + fell
39   [[the [horse [T race]]] [[past [the barn]] T]]
     [[the [horse [T race]]] [[past [the barn]] T]] + fell
40   [the horse]
     [the horse] + T/prt
41   [the [horse T/prt]]
     [the [horse T/prt]] + race
42   [the [horse T/prt(V)]]
     [the [horse T/prt(V)]] + past
43   [the [horse [T/prt(V) past]]]
     [the [horse [T/prt(V) past]]] + the
44   [the [horse [T/prt(V) [past the]]]]
     [the [horse [T/prt(V) [past the]]]] + barn
45   [the [horse [T/prt(V) [past [the barn]]]]]
     [the [horse [T/prt(V) [past [the barn]]]]] + T
46   [the [horse [T/prt(V) [past [[the barn] T]]]]]
     [the [horse [T/prt(V) [past [[the barn] T]]]]] + fell
47   [the [horse [T/prt(V) [past [the [barn T]]]]]]
     [the [horse [T/prt(V) [past [the [barn T]]]]]] + fell
48   [the [horse [T/prt(V) [[past [the barn]] T]]]]
     [the [horse [T/prt(V) [[past [the barn]] T]]]] + fell
49   [the [horse [[T/prt [race [past [the barn]]]] T]]]
     [the [horse [[T/prt [race [past [the barn]]]] T]]] + fell
50   [the [horse [[T/prt [race [past [the barn]]] [T fell]]]]] (<= accepted)
```

The exact derivation is sensitive to the actual parsing principles that are activated. For example, if we assume that the participle verb is activated before the finite verb (against experimental data), then the model will reach the correct solution without any garden paths.

There is only one situation in which incrementality is violated: inflectional features, as they are extracted from the phonological word, are stored in a temporary working memory and enter the syntactic component as stored inside a lexical item. The third person agreement marker -s in English, for example, will be put into a temporary memory hold, inserted inside the next lexical item (T), which then enters syntax. The element therefore stays in the memory only an extremely brief moment, being discharged always as soon as possible. In the case of several inflectional features, they are all stored in the same memory system and then inserted inside the next lexical item as a set of features (order is ignored).

3.3.2   Connectness

Connectness refers to the property that all incoming linguistic material is attached to one phrase structure that "connects" everything together. There never occurs a situation in which the syntactic working memory would hold two or more phrase structures that are not connected to each other by means of some grammatical dependency. It is important to keep in mind, though, that adjunct structures are attached to their host structures more loosely: they are geometrical constituents inside their host constructions, observing connectness, but invisible to many computational processes applied to the host (Section 2.7). We can imagine of them being "pulled out" from the computational pipeline processing the host structure and being processed by an independent computational sequence. Each adjunct, and more generally phase, is transferred independently and enters the LF-interface as an independent object.

3.3.3   Seriality

Seriality refers to the property that all operations of the parser are executed in a well-defined linear sequence. Although this is literally true of the algorithm itself, the detailed serial algorithmic implementation cannot be mapped directly into a cognitive theory for several reasons. One reason is that each linguistic computational operation performed during processing is associated with a *predicted cognitive cost*, measured in milliseconds, and it is the linear sum of these costs that provides the user the predicted cognitive processing time for each

word and sentence; CPU time is ignored. The current parsing model is serial in the sense that the predicted cognitive cost is computed in this way by adding the cognitive cost of each individual operation together, yet we could include parallel processing into the model by calculating the cognitive cost differently, i.e., not adding up the cost of computational operations that are predicted to be performed in parallel. The underlying implementation would still remain serial. Another complicating factor is that in some cases the implementation order does not seem to matter. We could simply *assume* that the processing is implemented by utilizing parallel processes. Despite these concerns, it is clear that most of the computational operations implemented by the linear phase model must be executed in a specific order in order to derive empirically correct results. Therefore, for the most part the model assumes that language processing is serial.

3.3.4   Locality preference

Locality preference is a heuristic principle of the human language comprehension system stating that local attachment solutions are preferred over nonlocal ones. A local attachment solution means the lowest right edge in the existing partial phrase structure representation. Thus, the preposition phase *with a telescope* in (55) is first attached to the lowest possible solution, and only if that solution fails, to the nonlocal node.

(55) John saw the girl with a telescope.

It is possible to run the parsing model with five different locality preference algorithms. They are as follows: *bottom-up*, in which the possible attachment nodes are ordered bottom-up; *top-down*, in which they are ordered in the opposite direction ('anti-locality preference'); *random*, in which the attachment order is completely random; *Z*, in which the order is bottom first, top second, then the rest in a bottom-up order; *sling*, which begins from the bottom node, then tries the top node and explores the rest in a top-down manner. The top-down and random algorithm constitute baseline controls that can be used to evaluate the efficiency of more realistic principles. The selected algorithm is defined for each independent study (Section 4.5.4). If no choice is provided, bottom-up algorithm is used as default.

3.3.5   Lexical anticipation

Lexical anticipation refers to parsing decisions that are made on the basis of lexical features. The linear phase parser uses several lexical features (Sections 2.4, 4.5.2). The system works by allowing each lexical feature to

vote each attachment site either positively or negatively, and the sum of the votes will be used to order the attachment sites. The weights, which can be zero, can be determined as study parameters (Section 4.5.4). This allows the researcher to determine the relative importance of various lexical features and, if required, knock them out completely (weight = 0). Large scale simulations have shown that lexical anticipation by both head-complement selection and head-specifier selection increases the efficiency of the algorithm considerably.

3.3.6   Left branch filter

The left branch filter closes parsing paths when the left branch constitutes an unrepairable fragment. The principle operates before other ranking principle are applied. The left branch filter can be turned on and off for each study (Section 4.5.4).

3.3.7   Working memory

There is substantial psycholinguistic literature that the operation of the human language comprehension module is restricted by a working memory bottleneck. After considerable amount of simulation and exploration of other models I concluded that this hypothesis could in fact be correct. The user can activate the working memory or knock it off by changing the study parameters in a manner explained in Section 4.5.4 and in this way see what its effects are.

In the current implementation the working memory operates in the following manner. Each constituent (node in the current phrase structure) is either active in the current working memory or inactive and out of the working memory. Any constituent β that arrives from the lexical component into syntax is active, and any complex constituent [α β] created thereby will be active. A constituent is *inactivated* and thus put out of the working memory when it is transferred and passes the LF-interface or when the attachment [α β] is rejected by ranking and/or by filtering; otherwise, it is kept in the working memory. It is assumed that a constituent residing out of the working memory is not processed in any way. Thus, all filtering and ranking principles cease to apply to it; it becomes passive. Finally, it is assumed that re-activation of a dormant constituent accrues a considerable cognitive cost, set to 500ms in this study. This implies that exploration of rejected parsing solutions will accrues much higher cognitive cost than they otherwise would do, as such dormant constituent must be reactivated into the working memory. This is due to the extra cognitive cost associated with the

reactivation but also due to the fact that ranking and filtering does not apply to such constituents, hence the system loses some 'grammatical intelligence' when it has to re-evaluate wrong parsing decisions made earlier.

3.3.8   Conflict resolution and weighting

The abovementioned principles may conflict. It is possible, for example, that locality preference and lexical anticipation provide conflicting results. Each possible conflict situation must be handled in some way. The conflict between locality preference and lexical anticipation is solved by assuming that locality preference defines the default behavior that is always outperformed by lexical anticipation. When different lexical features provide conflicting results, it is assumed that they cancel each out symmetrically. Thus, if head-complement selection feature votes against attachment [α β] but specifier selection favors it, then these votes cancel each other out, leaving the default locality preference algorithm (whichever algorithm is used). If two lexical features vote in favor, then the solution receives the same amount of votes as it would if only one positive feature would do the voting (+/− pair cancelling each other out, leaving one extra +). This result depends in how the various feature effects are weighted, which can again be provided independently for each study.

3.4   Measuring predicted cognitive cost of processing

Processing of words and whole sentences is associated with a predicted cognitive cost, measured in milliseconds. This is done by associating each word with a preprocessing time depending on its phonetic length (currently 25ms per phoneme) and then adding the predictive cognitive cost of each computational operation together. Most operations are currently set to consume 5ms, but the user can define these in a way that best agrees with experimental and neurobiological data. Reactivation of a constituent that is not inside the active working memory consumes 500ms. The resulting timing information will be visible in the log files and in the resource outputs. The processing time consumed by each sentence is simply the sum of the processing time of all of its words. A useful metric in assessing the relative processing difficulty of any given sentence is to calculate the mean predicted cognitive processing time per each word (total time / number of words). Currently the model predicts processing speeds of 600-700ms per word, much more if the processing involves garden paths. This metric takes sentence length into account. Resource consumption is summarized in the resource output file (Section 4.6.6) that lists each sentence together with the number of all computational operations

(e.g., Merge, Agree, Move) consumed in processing it from the reading of the first word to the outputting of the first legible solution. The file uses simple CSV format and can be read into an analysis program (Excel, SPSS, Matlab) or processed by using external Python libraries such as pandas or NumPy. The module *diagnostics.py* does this by using the latter. This makes it possible to examine the performance properties of various parsers and/or test corpuses and by doing this compare the results to human performance obtained under controlled conditions in the laboratory.

3.5   A note on implementation

Most of the performance properties are implemented in their own module *plausbility_metrics.py* which in essence determines how the attachment solutions (Section 2.3) are filtered and ordered. The abstract linear parser, which does not implement any performance properties by itself, sends all available attachment solutions to this module, which will first focus the operation to those nodes which are in the active working memory; the rest are not processed. The active nodes are then filtered, so that only valid solutions remain. The user can knock off all filters by changing the parameters of the study. The remaining nodes are then ordered by applying the selected locality preference algorithm, which provides the default ordering, and then by applying all other ranking principles such as lexical anticipation. Finally, the concatenated list of ordered nodes + nodes not active in the working memory and not processed are returned. The parser, which receives this list from the plausibility module, then uses this order in organizing its parsing derivation. Notice that the inactive nodes that are not in the active working memory must be part of the list as the parser must be able to explore them if everything else fails, but since the plausibility module does not process them, their order is independent of the incoming word and the partial phrase structure representation currently being constructed.

4   Inputs and outputs

4.1   Installation and basic use

The program is installed by cloning the whole directory from *Github*.

```
https://github.com/pajubrat/parser-grammar
```

The user must define a folder in the local computer where the program is cloned. This folder will then become the root folder for the project. The root folder will contain at least the following subfolders: *docs* (documentation, such as this focument), *language data working directory* (where each individual study is located) and *lpparse* (containing the actual Python modules). The software cannot currently be used via graphical user interface; it must be used by modifying the files with a text editor (such as Windows notepad) and by launching the program from the command prompt. In order to run the program the user must have Python (3.x) installed in the local computer and that installation must be specified in the windows path-variable. Refer to Python installation guide for how to accomplish these things on your local computer: the details depends on the operating system. The program can then be used by opening a command prompt into the program root folder and writing

```
python lpparse
```

into the command prompt, which will then parse all the sentences from the designated test corpus file in a study folder. This command will call the Python interpreter, as specified in the path-variable, and then feeds the *__main__.py* module from the folder */lpparse* into it. This will then call any other module, as required. The test corpus file and its location are provided in the *config.txt* file in the root directory that in my current local computer reads as

```
test_corpus_file:      linear_phase_theory_corpus.txt
test_corpus_folder:    study-6-linear-phase-theory
study_folder:          study-6-linear-phase-theory/
```

determining that in this case the sentences must be read from the file *linear_phase_theory_corpus.txt* in the folder *study-6-linear-phase-theory*.

4.2   General organization

The model was implemented and formalized as a Python 3.x program. It contains three main components. The first component is a *main script* responsible for performing and managing testing (*main.py*). 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.
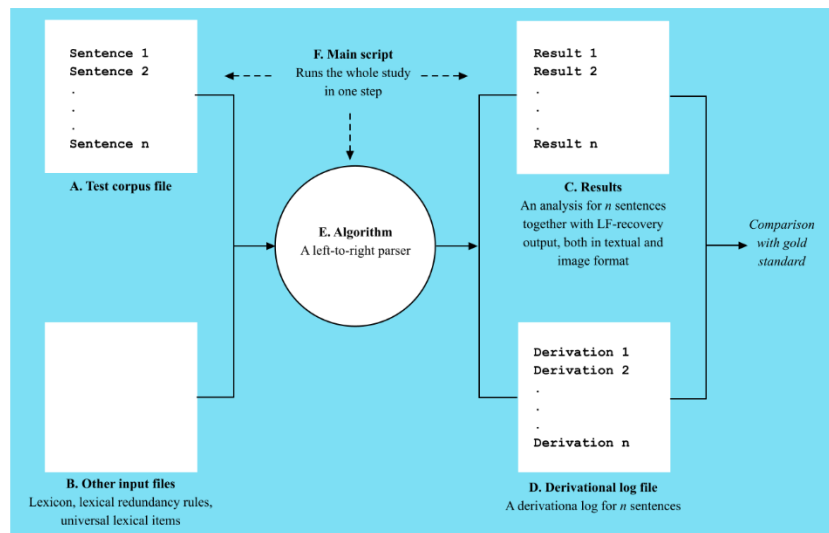
Figure 12. Relationships between the input, main script, linear phase parser and the output.

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 the process during the execution. The details of the operation of the main script will be elucidated further below.

The second component is the language comprehension module, 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 earlier in a nontechnical way. 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 program is contained in a directory structure that follows regular Python package conventions. The root directory has folds *docs* (documentation), *language data working directory* (all input and output files related to individual studies) and *lpparse* (program modules itself). The code exists in separate Python text files inside the folder *lpparse*. Individual modules, containing the program code, are ordinary *.py* text files that are all located in the master folder. The files and their contents are sketched in the table below.

Table. An alphabetical list of individual program modules

MODULE	DESCRIPTION
adjunct_constructor.py	This module processes externalization, in which a given element and/or the surrounding phrase is externalized, i.e. moved to the secondary system for independent processing. Linguistically it corresponds to the situation in which the element is promoted into an adjunct. It makes decisions concerning the amount of surrounding structure that will be externalized.
adjunct_reconstruction.py	Adjunct reconstruction takes place during transfer. It detects misplaced adjoinable phrases and reconstructs them by using tail features. It uses adjunct_constructor.py when needed.
agreement_reconstruction.py	Performs agreement reconstruction (Agree-1) for heads with +VAL.
diagnostics.py	Performs statistical analyses of the raw output data.
extraposition.py	Contains code that is used in extraposition, which refers to a type of externalization. Extraposition is attempted during transfer at two stages, first after head reconstruction and then as a last resort. In both cases, some fragment of the structure is externalized. During the first sweep, the system reacts to ungrammatical (and hence unrepairable) selection between reconstructed heads.
feature_disambiguation.py	This module performs feature manipulation (feature inheritance in the standard theory). It is currently solving the ambiguity with ?ARG feature and does nothing else, i.e. it sets the feature ?ARG into +ARG or -ARG. This is relevant for control. It is possible that its effects should be explained by lexical ambiguity.
knockouts.py	Contains metafunctions which allow the user to parametrize the model, i.e., to knockout various components of the algorithm.
head_reconstructiont.py	This module contains the code taking care of head reconstruction during transfer.
language_guesser.py	Hosts the code which determines the language used in an input sentence. The constructor will first read the lexicon and extract the languages available there, based on LANG features. The guesser will then

65

	determine the language of an input sentence on the basis of its words.
lexical_interface.py	This module reads and processes lexical information. Lexical information is read from the three external files and further processed through a function that applies "parameters". It is stored into a dictionary. Each parser object has its own lexicon that are initialized for each language in the main script.
LF.py	Processes LF-interface objects, with the main role being the checking of LF-legibility. It also hosts LFmerge operations, which are used in the production side to generate representations.
linear_phase_parser.py	This module defines the parser and its operations (main parse function plus the recursive function called by the former). This is a purely performance module: it reads the input sentence, generates the recursive parse tree, evaluates and stored the results. Ranking is currently inside this module, but it will be moved out later into its own module.
log_functions.py	Contains two logging related operations, one which logs the sentence before parsing and another which stores the results.
main.py	Function that is executed when the user launches the program from the command prompt. This function interprets command line arguments and prepares the study accordingly.
morphology.py	Contains code handling morphological processing, such as morphological decomposition and application of the mirror principle. This module uses only linear representations.
multistudy.py	List of functions that allows one to run several studies at once.
parse.py	Main script. This script is written as a linear sequence of commands that prepare the parsers (for each language), sends all input sentences into the appropriate parser and stores the results.
phrasal_reconstruction.py	Contains code implementing phrasal reconstruction, both A-bar reconstruction and A-reconstruction. These operations are part of transfer.
phrase_structure.py	A class that defines the phrase structure objects and the grammatical configurations and relations defined on them.
semantics.py	This module interprets the output of the parser semantically.
support.py	Contains various support functions that are irrelevant to the empirical model itself.
surface_conditions.py	The module contains filters that are applied to the spellout structure. In the current implementation it only contains tests for incorporation integrity that are used in connection with clitic processing. It will contain also functions pertaining to surface scope.
transfer.py	This module performs the transfer operation. It contains a list of subprocesses in a specific order of execution (head reconstruction, feature processing, extraposition, adjunct reconstruction, phrasal reconstruction, agreement reconstruction, last resort extraposition)

| visualizer.py | Hosts the code used to generate images of phrase structure trees. |

The program is run by writing command *python lpparse* into the command prompt in the root folder of the program. This command will then launch the function *__main__.py* inside the *lpparse* folder. The purpose of this function is to direct the execution of the study on the basis of the parameters provided by the user. Currently the following uses are possible:

```
python lpparse

python lpparse diagnostics

python lpparse multi
```

The first will launch one specific study. The second will run statistical analyses on the output files of studies that have already been run. Finally, the third runs several studies in one go. Suppose we want to adopt the first option and run only one study. The location of that study (input and output files) is specified in the file *config.txt* located in the root directory. It may contain, for example, the following three lines:

```
test_corpus_file:      linear_phase_theory_corpus.txt
test_corpus_folder:    study-6-linear-phase-theory
study_folder:          study-6-linear-phase-theory/
```

These lines tell the program that it should process sentences from the test corpus file named *linear_phase_theory_corpus.txt* that is located in the folder *study-6-linear-phase-theory*. The last line will tell the program that all other study-related material (input, output) will be generated into the folder *study-6-linear-phase-theory*. This folder and the test corpus folder are typically the same, but they do not need to be the same.

The script that runs one study is *main.py*. This 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 in the folder *lpparse* where the main script is. The *config.txt* file is read by *main.py* module. Once the *main.py* knows the location of the test corpus and the study parameters, it will read them. Individual study parameters are provided in the designated study folder in a file *config_study.txt*. We will examine the content of that file later.
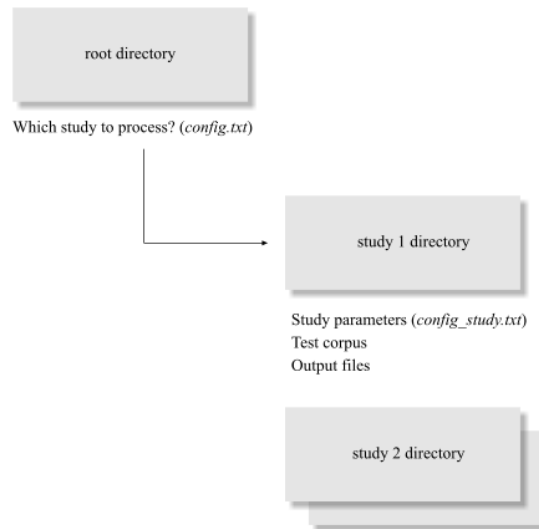
Figure 31. General organization. The main function *main.py* locates the *config.txt* from the root directory which it uses to direct its attention to the study parameters.

The motivation for this two-layer architecture is that we want to be able to run several different studies, each with possible different test corpora and test parameters. The higher-level configuration file *config.txt* in the root directory tells what study we want to run by specifying its folder and the test corpus file. We put all study material inside that folder. The second-level configuration file *config_study.txt* inside the study folder then tells what parameters we use in that particular study. These properties will later be replaced by a graphical interface (GUI) which will make the system hopefully much easier to use.

Individual studies are associated with specific input files inside the *language data working directory* subfolder, which contains a further subfolder for each study, published, submitted or in preparation. A copy of each lexical file exists also in the language data working directly, which makes it possible to work with one "master lexicon." If the lexical files existed only inside individual study folders, then there will be parallel lexicons that differ from each other, making it harder to keep track of the overall development. Once a study is published, however, a copy of the lexical resources used in that study should be stored in connection with the rest of the study-specific materials inside the specific folder.

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. The program branches are also 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.

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 6.x, the version associated with the present document, refers to a version have been implemented after study 6. The current version is 6.9, thus very close to the submission of the study number 7. 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 2020c); 3 = control and null arguments (Brattico 2020d); 4 = head movement reconstruction and Finnish long head movement (Brattico 2020e); 5 = first study of clitics and incorporation (Brattico 2020b); 6 = case marking study (submitted); 7 = performance study (in preparation).

4.3   Running several studies

It is possible to run several studies at once. This can be done by write command *python lpparse multi* at the command line. The argument *multi* will execute the module *multistudy.*py which contains simply a list of commands that are used to run individual studies. Each individual study is configured by providing the study configuration (i.e., test corpus, test corpus location, study location) as function arguments, not as an external file. In other words, instead of reading the configuration from the external file config.txt, the same information is provided as arguments to the parser function (*main.run_study* function). See the *multistudy.py* module itself.

One important use case for the multistudy functionality is that if a published study requires that we process several test corpora with possibly different parameters, then writing the whole study in the form of a one meta-script allows one to repeat and replicate the exact same study.

4.4 Main script (*main.py*)

The main script runs one individual study on the basis of parameters provided in the study folder as defined in the root directory file *config.txt* or as provided by arguments (see 4.3). It configures the parser on the basis of the study parameters, provided in the same folder, and the feeds all sentences from the test corpus into the parser and records the outputs into separate files. The output is stored into the study folder.

4.5 Structure of the input files

4.5.1 Test corpus file (any name)

The test corpus file name and location are provided in the *config.txt*. 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. 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.

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 it will appear in the results file as well. This allows the user to leave comments into the results file that would otherwise get populated with raw data only. Should the user want to group the sentences by using numerical coding, this is possible by writing =>x.y.z.a, for example =>1.1.1.0. This will label all following sentences with that numerical code, until another similar line occurs. These numbers are very useful if we want later to analyze the results on the basis of some grouping scheme. 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 a group of sentences, they should all be prefaced with + symbol. The rest of the sentences are then ignored. 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. To being processing in the middle of the file, use

the symbol =START= (in effect, sentences between =START= and =STOP= still be processed).[6] Figure 14 is

a screen capture from one test corpus file to illustrate what it looks like.
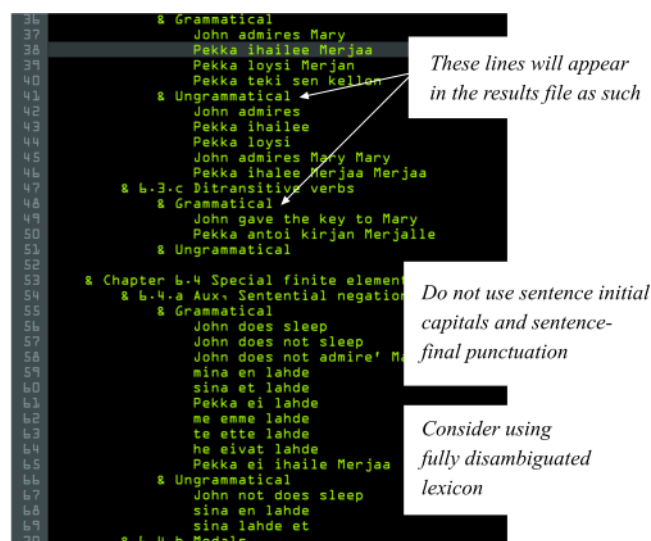
Figure 14. Screen capture of a test corpus file.

4.5.2   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.

---

[6] It is possible to use several =START= and =STOP= commands. They are interpreted so that all previous items are disregarded each time =START= is encountered, whereas =STOP= disregards anything that follows. Thus, only the last =START= and the first =STOP will have an effect.

```
 5 admire   :: admire-#v#T/fin#[-0] LANG:EN
 6 admire'  :: admire-#v LANG:EN
 7 admires  :: admire-#v#T/fin#[-s] LANG:EN
 8 admire-  :: PF:admire LF:admire V CLASS:TR -SPEC:Neg -COMP:Neg COMP:D
 9
10 adoro    :: adora-#v#T/fin#[-o] LANG:IT
11 adori    :: adora-#v#T/fin#[-i] LANG:IT
12 adora    :: adora-#v#T/fin#[-a] LANG:IT
13 adoriamo :: adora-#v#T/fin#[-iamo] LANG:IT      Morphological
14 adorate  :: adora-#v#T/fin#[-te] LANG:IT        decomposition
15 adorano  :: adora-#v#T/fin#[-no] LANG:IT
16 adora-   :: PF:adora LF:admire V COMP:D LANG:IT
17
18 anta-    :: PF:antaa LF:give V CLASS:DITR -COMP:FIN +SEM:directional LANG:FI
19 antoi    :: anta-#v#T/fin#[-V] LANG:FI
20
21 asks     :: ask-#v#T/fin#[-s] LANG:EN
22 ask'     :: PF:ask LF:ask V SPEC:D COMP:D SEM:internal LANG:EN
23 ask-     :: PF:ask LF:ask V SPEC:D COMP:D SEM:internal LANG:EN
24
25 avain_Dacc  ::  avain-#D#[-D_acc]
26 avain_nom   ::  avain-#D#[-D_nom]            List of features associated
27 avain       ::  avain-#D#[-nom]             with a primitive item
28 avaimen_acc ::  avain-#D#[-n_acc]
29 avaimen     ::  avain-#D#[-n_acc]
30 avaimet     ::  avain-#D#[-t_acc]#pl
31 avain-   ::  PF:avain LF:key N LANG:FI -SEM:directional
32
33 auton    ::  auto-#D#[-n_acc] LANG:FI
34 auto     ::  auto-#D LANG:FI
35 auto-    ::  PF:auto LF:car N LANG:FI -SEM:directional
36
37 city     :: PF:city LF:city N LANG:EN
38
39 detesto  :: detest-#v#T/fin#[-o] LANG:IT
```

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.[7] 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.

LF:__	Semantic access key, concept, meaning, mental image	!SPEC:__	Label of a mandatory specifier
PF:__	Phonological form, surface form	-SPEC:__	Label of an impossible specifier
__	Lexical category (e.g. V, N, A)	SEM:__	Semantic feature
LANG:__	Language	TAIL:__,__	Tail-head set
COMP:__	Label of an acceptable complement	+/-ARG	Presence/absense of unvalued phi
!COMP:__	Label of a mandatory complement	+/-VAL	Presence/absense of valuation (Agree-1)
-COMP:__	Label of an impossible complement	PHI:__:__	Phi-feature of type __ with value __
SPEC:__	Label of an acceptable specifier	ASP	Aspectual head which projects it own event and thematic structure (will have valued in future implementations)

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 "–" or by the word "inflectional". 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.

---

[7] They are currently implemented by simple string operations, but in some later iteration all such processing will be replaced by regex processing.

4.5.3 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 or antecedent 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.


```
 3
 4 # Basic lexical categories
 5 FORCE   :: -ARG -VAL FIN !COMP:* -SPEC:* !PROBE:T/fin COMP:T/fin COMP:C/fin
 6 C/fin   :: -ARG VAL FIN C SPEC:* !COMP:* -COMP:C/fin -COMP:T/prt COMP:T/fin !PROBE:FIN -SPEC:T/f
 7 T/fin   :: ARG VAL T FIN !COMP:* COMP:T/prt COMP:v COMP:V COMP:D !PROBE:V SPEC:D -SPEC:N -SPEC:T
 8 T/prt   :: ARG VAL T FIN !COMP:* COMP:v COMP:V COMP:T/prt -COMP:NEG !PROBE:V SPEC:D -SPEC:N -SPE
 9 T       :: ARG VAL FIN !COMP:* COMP:v COMP:V COMP:T/prt -COMP:NEG !PROBE:V SPEC:D -SPEC:N -SPEC:
10 Neg/fin :: ARG VAL FIN NEG COMP:T/prt -SPEC:T -SPEC:T/fin %SPEC:INF SEM:internal
11 v       :: ARG -VAL ASP !COMP:* COMP:V !PROBE:V SPEC:D -SPEC:N
12 V       :: ARG -VAL ASP -SPEC:T/fin -SPEC:FORCE SPEC:ADV -COMP:N -COMP:T -COMP:V -COMP:v SPEC:P
13 D       :: -ARG VAL OP !COMP:* -COMP:D -SPEC:MA/lla -SPEC:P -SPEC:N -SPEC:INF -SPEC:D -SPEC:V CO
14 N       :: -COMP:P -COMP:A -COMP:AUX -SPEC:FORCE -COMP:C/fin SPEC:A COMP:R COMP:R/D -COMP:D -COM
15 P       :: ARG -VAL !COMP:* !COMP:D -COMP:N -COMP:ADV -COMP:T/fin -SPEC:iR -SPEC:iWH -SPEC:C/fin
16 ADV     :: -SPEC:N -SPEC:FORCE -SPEC:Neg/fin -SPEC:T/fin adjoinable
17 A       :: COMP:D TAIL:D -SPEC:A adjoinable
18 INF     :: !COMP:* COMP:v COMP:V !PROBE:V -COMP:FORCE -COMP:C/fin -SPEC:T/fin -SPEC:V
19 n       :: ARG VAL COMP:D COMP:V COMP:v SPEC:*
20 D       :: SPEC:D 2SPEC
```

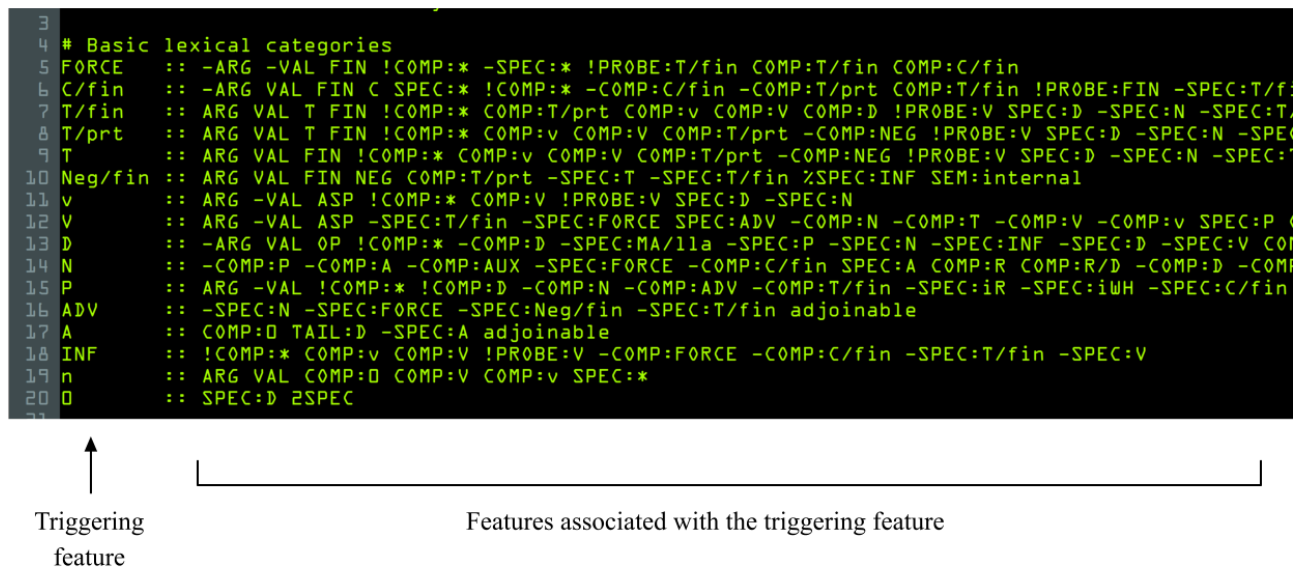Triggering   Features associated with the triggering feature
feature

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'. It is also possible (and in some cases needed) to use language specific redundancy rules. These are represented by pairing the antecedent feature with a language feature (e.g., LANG:FI).

4.5.4 Study parameters (*config_study.txt*)

It is possible to associate each study with specific study parameters which tell how the parser operates. These parameters are contained in the file *config_study.txt*. The parameters are also stamped on the output files, so that it is possible later to examine what parameters were used in running the study. Here is a list of the parameters currently in use:

```
Author, year, date = author, year and date of the study.
study_id = the numerical id number of the study (0, 1, 2, ...)
logging = whether logging is allowed or blocked (True, False)
ignore_ungrammatical_sentences = whether ungrammatical sentences are processed (True, False)
datatake_resource = whether resource data is generated (True, False)
datatake_resource_sequence = whether a resource sequence is generated (True, False)
datatake_timings = whether timing data is generated (True, False)
datatake_images = whether phrase structure images are generated (True, False)
image_parameters...= several parameters considering the images, if generated
extra_ranking = whether extra ranking principles are used (True, False)
filter = whether left branch filter is used (True, False)
lexical_anticipation = whether lexical anticipation principle is used (True, False)
closure = the type of locality preference (bottom-up, top-down, random, Z, sling)
working_memory = whether working memory bottleneck is active (True, False)
```

This is followed by numerical weights for various lexical anticipation principles which allows very detailed tuning of the parser.

4.6 Structure of the output files

4.6.1 Results

The name and location of the results output file is determined when configuring the main script, either in the external file *config.txt* or by arguments when using the multi study functionality. 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. 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.

```
  1 2020-05-23 09:36:03.815284
  2 Test sentences from file "language data working directory\study-3_2020-control\null_subjects_corpus.t>
  3 Logs into file "language data working directory\study-3_2020-control\null_subjects_corpus_log.txt.
  4 Lexicon from file "language data working directory\study-3_2020-control\lexicon.txt".
  5 & Group 0.1 Example equivalents from the main article ----------------------------------------------
  6
  7 & Example 1 --------------------------------------------------
  8
  9 1. John wants to_inf leave
 10
 11     [[D John]:1 [T/fin [__:1 [v [want [to leave]]]]]]
 12     LF_Recovery:
 13     Agent of leave(John)
 14     Agent of to(John)
 15     Agent of v(John)
 16     Agent of want(John)
 17
 18 2. John wants Mary to_inf leave
 19
 20     a. [[D John]:1 [T/fin [__:1 [v [want [[D Mary]:2 [to [__:2 leave]]]]]]]]
 21     LF_Recovery:
 22     Agent of leave(Mary)
 23     Agent of v(John)
 24     Agent of want(John)
 25
 26     b. [[[D John]:1 [T/fin [__:1 [v [want [D Mary]]]]]] <to leave>]
 27     LF_Recovery:
```

Timestamp and locations of the input files

Results

Sentence from the test corpus file with numerical identifier (# 1) provided by the main script

Figure 19. Screen capture from a results file.

The algorithm stores grammaticality judgements into a separate file names "_grammaticality_judgemnts.txt", which contains the groups, numbers, sentences and grammatical judgments. This is useful if you have a voluminous test corpus and want to evaluate it very efficiently. To do this, first use the same format to create gold standard by using native speaker input, store that data with a separate name, and then compare the algorithm output with the gold standard by using automatic comparison tools.

4.6.2   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).

```
 2
 3 \==================================================================
 4 # 1
 5 ['John', 'wants', 'to_inf', 'leave']
 6 Using lexicon "language data working directory\study-3_2020-control\lexicon.txt".
 7
 8          =None
 9
10      Next word contains multiple morphemes ['m$', 'hum$', 'def$', '3p$', 'sg$', 'D$', 'John-']
11      Storing inflectional feature ['-', 'LANG:EN', 'PHI:GEN:M'] into working memory.
12
13   1. Consume "hum$"
14      Storing inflectional feature ['-', 'LANG:EN', 'PHI:HUM:HUM'] into working memory.
15
16   2. Consume "def$"
17      Storing inflectional feature ['-', 'LANG:EN', 'PHI:DET:DEF'] into working memory.
18
19   3. Consume "3p$"
20      Storing inflectional feature ['-', 'LANG:EN', 'PHI:PER:3'] into working memory.
21
22   4. Consume "sg$"
23      Storing inflectional feature ['-', 'LANG:EN', 'PHI:NUM:SG'] into working memory.
24
25   5. Consume "D$"
26      Adding inflectional features {'LANG:EN', 'PHI:HUM:HUM', 'PHI:GEN:M', 'PHI:NUM:SG', 'PHI:PER:3', '-',
27          =D
28
29   7. Consume "John"
30
31      D + John
32          Filtering out impossible merge sites...
33          Sink "John" into D because they are inside the same phonological word.
34          =D{N}
35
36      Next word contains multiple morphemes ['[-s]$', 'T/fin$', 'v$', 'want-']
37      Storing inflectional feature ['-', 'LANG:EN', 'PHI:GEN:F', 'PHI:GEN:M', 'PHI:NUM:SG', 'PHI:PER:3'] in
38
```

1

2

3

4

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.

```
137
138      >>> Trying candidate spell out structure [[D John] [T/fin{v,V} [to leave    1
139         Checking surface conditions...
140         Reconstructing...
141             1. Head movement reconstruction:
142                 Target v{V} in T/fin
143                 =[[D John] [T/fin [v{V} [to leave]]]]
144                 Target want in v
145                 =[[D John] [T/fin [v [want [to leave]]]]]
146             =[[D John] [T/fin [v [want [to leave]]]]]
147             2. Feature processing:
148                 Solving feature ambiguities for "to".
149             =[[D John] [T/fin [v [want [to leave]]]]]
150             3. Extraposition:
151             =[[D John] [T/fin [v [want [to leave]]]]]         2
152             4. Floater movement reconstruction:
153             =[[D John] [T/fin [v [want [to leave]]]]]
154             5. Phrasal movement reconstruction:
155                 [D John] will undergo A-reconstruction.
156             =[[D John]:4 [T/fin [__:4 [v [want [to leave]]]]]]
157             6. Agreement reconstruction:
158                 Head T/fin triggers Agree-1:
159                     T/fin acquired PHI:GEN:M by phi-Agree from __:4.
160                     T/fin acquired PHI:NUM:SG by phi-Agree from __:4.
161                     T/fin acquired PHI:PER:3 by phi-Agree from __:4.
162                     T/fin acquired PHI:DET:DEF from the edge of T/fin.
163                 Head to triggers Agree-1:
164             =[[D John]:4 [T/fin [__:4 [v [want [to leave]]]]]]
165             7. Last resort extraposition:
166           = [[D John] [T/fin [[D John] [v [want [to leave]]]]]]
```

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 =.


```
167         Checking LF-interface conditions.
168             Transferring [[D John]:4 [T/fin [__:4 [v [want [to leave]]]]]] into the conceptual-intentional system...
169                 v with ['PHI:DET:_', 'PHI:NUM:_', 'PHI:PER:_'] was associated at LF with:
170                     1. [D John]    (alternatives: 2. T/fin )
171                 want with ['PHI:DET:_', 'PHI:NUM:_', 'PHI:PER:_'] was associated at LF with:
172                     1. [D John]    (alternatives: 2. T/fin )                                  1
173                 to with ['PHI:NUM:_', 'PHI:PER:_'] was associated at LF with:
174                     1. [D John]    (alternatives: 2. T/fin )
175                 leave with ['PHI:DET:_', 'PHI:NUM:_', 'PHI:PER:_'] was associated at LF with:
176                     1. [D John]    (alternatives: 2. T/fin )
177             Transfer to C-I successful.
178             Semantic interpretation/predicates and arguments: [' ', 'Agent of leave(John)', 'Agent of to(John)', 'Agent of v(J
179 -------------------------------------------------------------------------------------
180                                     All tests passed
181 -------------------------------------------------------------------------------------
182     Solution:
183         [[D John] [T/fin [[D John] [v [want [to leave]]]]]]                               2
184         Grammar: [[D John]:1 [T/fin [__:1 [v [want [to leave]]]]]]
185         Spellout TT/finP = [DP:1 [TT/fin [__:1 [v [V [INF V]]]]]]
186 -------------------------------------------------------------------------------------
187 D:['!COMP:*', '!PROBE:CAT:N', '-', '-ARG', '-COMP:T/fin', '-COMP:uR', '-SPEC:D', '-SPEC:N', '-SPEC:Neg/fin', '-SPEC:T/fin', '-SPEC
188 John:['-COMP:ADV', '-COMP:AUX', '-COMP:D', '-COMP:N', '-COMP:P', '-COMP:T/fin', '-COMP:V', '-COMP:WH', '-COMP:v', '-SEM:directiona
189 T/fin:['!COMP:*', '!PROBE:CAT:V', '!SPEC:*', '-', '-SPEC:FORCE', '-SPEC:N', '-SPEC:T/fin', '-SPEC:V', 'ARG', 'ASP', 'CAT:ARG', 'CA
190 D:['!COMP:*', '!PROBE:CAT:N', '-', '-ARG', '-COMP:T/fin', '-COMP:uR', '-SPEC:D', '-SPEC:N', '-SPEC:Neg/fin', '-SPE     N', '-SPEC
191 John:['-COMP:ADV', '-COMP:AUX', '-COMP:D', '-COMP:N', '-COMP:P', '-COMP:T/fin', '-COMP:V', '-COMP:WH', '-COMP:v',          irectiona   3
192 v:['!COMP:*', '!PROBE:CAT:V', '!SPEC:D', '-SPEC:N', '-VAL', 'ARG', 'ASP', 'CAT:ARG', 'CAT:ARG/v', 'CAT:v', 'COMP:\    G:EN', 'L
193 want:['!COMP:*', '-COMP:ADV', '-COMP:N', '-COMP:T/fin', '-COMP:V', '-SPEC:FORCE', '-SPEC:T/fin', '-SPEC:TO/inf', '-V   ', 'ARG', 'A
194 to:['!COMP:*', '!PROBE:CAT:V', '-COMP:C/fin', '-COMP:FORCE', '-COMP:T/fin', '-SPEC:T/fin', '-SPEC:V', '?VAL', 'ARG', 'ASP', 'CAT:A
195 leave:['!SPEC:D', '-COMP:ADV', '-COMP:N', '-COMP:T/fin', '-COMP:TO/inf', '-COMP:V', '-SPEC:FORCE', '-SPEC:T/fin', '-VAL', 'ARG', '
196
```

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.

4.6.3 Simple logging

A file ending with *_simple_log.txt* contains a highly simplified log file which shows only a list of the partial phrase structure representations and accepted solutions generated during the derivation. This is very useful if the user wants to see with one glance what the parser did.

4.6.4 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. 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 4.6.2) and does not appear in this file.

4.6.5 Images of the phrase structure trees

The algorithm stores the parsing output in phrase structure images (PNG format) if the user activates the corresponding functionality. The function can be activated by input parameters in the study configuration file (Section 4.5.4). Figure 23 illustrates the phrase structure representation generated for a simple transitive clause in English, when produced without any lexical information.
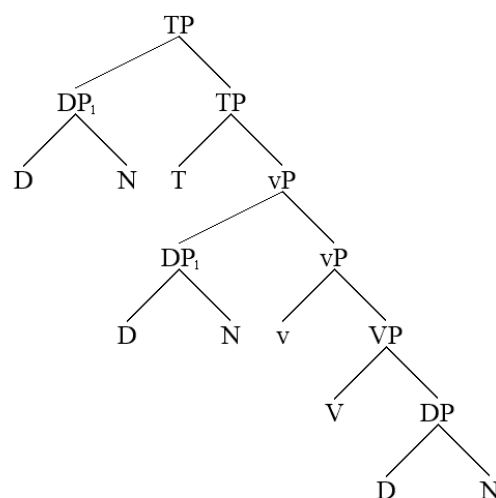
Figure 23. A simple phrase structure image generated by the algorithm for a simple transitive clause *John admires Mary*.

The lexical category labels shown in these images 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).

As pointed out above, it is possible to add lexical information to the primitive items. This is useful tool when examining the output but overlapping text may sometimes create unappealing visualizations. In that case, the user might want to edit the figure manually. To do this, first activate the slow mode (parameter /slow) which halts the processing after each image. The user can edit the image while it is displayed on the screen. You can select a node in the tree by using the mouse, and then move it either by using cursor keys or by dragging with a mouse. Pressing 'R' will reset the image. Once you have done all required edits, press 'S' to save the image and close the window to proceed to the next image. If the user wants to add textual fields or other ornamentation to the image, this should be done in a separate program (such as Adobe Illustrator). If you close the image without saving, it will not be saved.

4.6.6 Resources file

The algorithm records the number of computational operations and CPU resources (in milliseconds) consumed during the processing of the first solution. At the present writing, these data are available only for the first solution discovered. Recursive and exhaustive backtracking after the first solution has been found, corresponding to a real-world situation in which the hearer is trying to find alternative parses for a sentence that has been interpreted, is not relevant or psycholinguistically realistic to merit detailed resource reporting. These processed are included in the model only to verify that the parser is operating with the correct notion of competence and does not find spurious solutions. In addition, resource consumption is not reported for ungrammatical sentences, as they always involve exhaustive search.

Resource consumption is reported in two places. A summary is normally provided in the results file. In addition, the algorithm generates a file with the suffix "_resources" to the study folder that reports the results in a format that can be opened and processed directly with external programs, such as MS Excel or by using

external Python libraries such as pandas or NumPy. To see an example, see the *diagnostics.py* module. The list of resources reported is provided in the parser class and can be modified there. In addition to listing resource consumption, each line contains also the study number (as specified in the input parameters) and the numerical classifications read from the test corpus file, if any (see Section 4.5.1).

4.6.7   How to add own log entries

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

5   Grammar formalization

5.1   Basic grammatical notions (phrase_structure.py)

5.1.1   Introduction

The class PhraseStructure (defined in *phrase_structure.py*) defines the phrase structure objects called

constituents that are manipulated at each stage of the processing pipeline.

5.1.2   Types of phrase structure constituents

5.1.2.1   Primitive and terminal constituents

A constituent is *primitive* if and only if it does not have both the left and right constituent.

```
def is_primitive(self):
    return not (self.right_const and self.left_const)
```

It follows that a constituent that has zero or one constituents is primitive. A *terminal constituent* is one that

has no constituents. They are made up of *lexical items*, which are sets of features.

```
def terminal_node(self):
    return self.is_primitive() and not self.has_affix()
```

5.1.2.2   Complex constituents; left and right daughters

A constituent is *complex* if and only if it is not primitive.

```
def is_complex(self):
    return not self.is_primitive()
```

It follows that a complex constituent must have a *left constituent* and a *right constituent*. These notions are

defined as follows.

```
def is_left(self):
    return self.mother and self.mother.left_const == self
```

```
def is_right(self):
    return self.mother and self.mother.right_const == self
```

The phrase structure tree made up of constituents is binary branching. Some derivative definitions that are used to simplify the code in certain places are as follows:

```
def left_primitive(self):
    return self.left_const and self.left_const.is_primitive()

def left_complex(self):
    return self.left_const and self.is_complex()

def bottom(self):
    while not self.is_primitive():
        self = self.right_const
    return self

def contains_feature(self, feature):
    if self.left_const and self.left_const.contains_feature(feature):
        return True
    if self.right_const and self.right_const.contains_feature(feature):
        return True
    if self.is_primitive:
        if feature in self.features:
            return True
    return False
```

5.1.2.3   Complex heads and affixes

A constituent constitutes a *complex head* and contains one or several internal parts if and only if it has the right constituent but not the left constituent. The orphan right constituent will hold an internal morpheme. Notice that a complex head is a primitive constituent despite containing a constituent.

```
def has_affix(self):
    return self.right_const and not self.left_const


def is_complex_head(self):
    return self.is_primitive() and self.has_affix()


def get_affix_list(self):
    lst = [self]
    while self.right_const and not self.left_const:
        lst.append(self.right_const)
        self = self.right_const
    return lst


def get_affix_list(self):
    lst = [self]
    while self.right_const and not self.left_const:
        lst.append(self.right_const)
        self = self.right_const
    return lst


def bottom_affix(self):
    if not self.is_primitive:
        return None
    ps_ = self
    while ps_.right_const:
        ps_ = ps_.right_const
    return ps_
```

Recall that we defined *terminal node* as a constituent that is primitive but does not have any affix.

```
def terminal_node(self):
    return self.is_primitive() and not self.has_affix()
```

5.1.2.4 Externalization and visibility

Adjuncts are *externalized*, and those constituents that have not been externalized are also called *visible*.

```
def externalized(self):
    return self.adjunct

def visible(self):
    return not self.externalized()

def adjoinable(self):
    return 'adjoinable' in self.features and '-adjoinable' not in self.features

def is_adjoinable(self):
    return self.externalized() or 'adjoinable' in self.head().features
```

5.1.2.5 Sisters

Two constituents are *geometrical sisters* if they occur inside the same constituent. The right constituent constitutes the geometrical sister of the left constituent, and vice versa.

```
def geometrical_sister(self):
    if self.is_left():
        return self.mother.right_const
    if self.is_right():
        return self.mother.left_const
```

The notion of geometrical sister refers to a relation of sisterhood that is defined purely in terms of phrase structure geometry. We will often use a narrower notion, called *sister*, that ignores externalized right adjuncts.

```
def sister(self):
    while self.mother:
        if self.is_left():
            if self.geometrical_sister().visible():
                return self.geometrical_sister()
            else:
                self = self.mother
        if self.is_right():
            if self.visible():
                return self.geometrical_sister()
            else:
                return None
    return None
```

This definition introduces two extra conditions: the right sister of a left sister must be visible, and the left constituent of a right constituent can constitute its sister only if the right constituent is visible. In effect, the definition ignores invisible right constituents. We will examine the meaning of this definition later. The fact that sisterhood relation is defined in this way is extremely important.

5.1.2.6 Proper complement and complement

A *proper complement* of a constituent is its right sister.

```
def proper_complement(self):
    if self.sister() and self.sister().is_right():
        return self.sister()
```

84

A (regular) *complement* is defined as the same relation as sisterhood. Geometrical sisterhood plays no relation is complement. We use also a host of derivative definitions based on lexical features (attribute *features* that refers to a set).

```
def missing_complement(self):
    return self.is_primitive() and not self.proper_complement() and self.licensed_complements()

def wrong_complement(self):
    return self.is_left() and self.proper_complement() and self.has_mismatching_complement()

def complement_match(self, const):
    return self.licensed_complements() & const.head().features

def licensed_complements(self):
    return {f[5:] for f in self.features if f[:4] == 'COMP'} | {f[6:] for f in self.features if f[:5] == '!COMP'}

def complements_not_licensed(self):
    return {f[6:] for f in self.features if f[:5] == '-COMP'}

def has_mismatching_complement(self):
    return not (self.licensed_complements() & self.proper_complement().head().features)

def get_mandatory_comps(self):
    return {f[6:] for f in self.features if f[:5] == '!COMP' and f != '!COMP:*'}
```

5.1.2.7   Labels

Labeling algorithm is based on (56).

(56) 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).

```
def head(self):
    if self.is_primitive():
        return self
    if self.left_const.is_primitive():
        return self.left_const
    if self.right_const.is_primitive():
        return self.right_const
    if self.right_const.externalized():
        return self.left_const.head()
    return self.right_const.head()
```

Labeling allows us to define *get_max*.

```
def get_max(self):
    ps_ = self
    while ps_ and ps_.mother and ps_.mother.head() == self.head():
        ps_ = ps_.walk_upstream()
    return ps_
```

5.1.2.8   Minimal search, geometrical minimal search and upstream search

Several operations require that the phrase structure is explored in a pre-determined order. In a typical scenario, minimal search from α explores the right edge of α in a downstream direction: left branches are phases and are not visited. Thus, minimal search from [α β] goes into β. The operation of right edge exploration would be trivial to define were there no right-adjuncts; thus, right-adjuncts are externalized and hence ignored when exploring the right edge (by definition of "right edge," which applies to current working space only). Thus, the right edge of [α, β] is β, but the right edge of [α, $\langle\beta\rangle$] is α. (There is a separate notion, geometrical minimal search that is used in parsing, which does not ignore adjuncts.) The following terminology is consistently used in the code and in the publications. *Downstream* is defined by downstream search, thus in a typical case it denotes β in [$_{\beta P}$ α β]. *Geometrical downstream* always denotes β in [α β]. *Leftward* is defined by the left constituent when downstream points to the right constituent (e.g., α in [$_{\beta P}$ α β]). *Rightward* is defined by the right constituent when downstream points to the left constituent (e.g., β in [$_{\alpha P}$ α, $\langle\beta\rangle$]). *Upward* denotes the mother constituent (e.g., [α β] for α, β).

Minimal search on α constitutes an operation that crawls downwards on the right edge of α. It is defined by creating an iteration over phrase structure that is then used in a simple *for* loop. Thus, the definition for minimal search itself is given as follows.

```
def minimal_search(self):
    return ⟦node for node in self⟧
```

The more important part is contained in the definition for *__getitem__* function that allows us to enumerate the constituents of a phrase structure.

```
def __getitem__(self, position):
    iterator_ = 0
    ps_ = self
    while ps_:
        if iterator_ == position:
            return ps_
        if ps_.is_primitive():
            raise IndexError
        else:
            if ps_.head() == ps_.right_const.head():    # ⟦_YP XP YP⟧ = YP
                ps_ = ps_.right_const
            else:
                if ps_.left_const.is_complex():          # ⟦_XP XP <YP>⟧ = XP
                    ps_ = ps_.left_const
                else:
                    if ps_.right_const.externalized():   # ⟦X <YP>⟧ = X
                        ps_ = ps_.left_const
                    else:
                        ps_ = ps_.right_const            # ⟦X Y⟧ = Y
        iterator_ = iterator_ + 1
```

86

Geometrical minimal search depends on the phrase structure geometry alone and does not respect labelling and visibility.

```
def geometrical_minimal_search(self):
    search_list = [self]
    while self.is_complex() and self.right_const:
        search_list.append(self.right_const)
        self = self.right_const
    return search_list
```

An *upstream search* is based on dominance and motherhood relation.

```
def upstream_search(self):
    path = []
    while self.mother:
        path.append(self.mother)
        self = self.mother
    return path
```

5.1.2.9   Edge and local edge

The *edge* of a constituent contains left complex constituents inside its own projection together with certain features of the head itself.

```
def edge(self):
    edge = []
    # ------------- minimal upstream path ---------------------#
    for node in self.upstream_search():
        if node.head() != self:
            break
        if node.left_const and node.left_const.is_complex() and node.left_const.head() != self:
            edge.append(node.left_const)
    #--------------------------------------------------------#
    if not edge and self.extract_pro():
        edge.append(self.extract_pro())
    return edge
```

This definition implies that if no complex left phrases are present, the edge may contain pro-elements that it reconstructs from the phi-features of the head. *Local edge* is defined as the first element in the edge.

```
def local_edge(self):
    if self.edge():
        return self.edge()[0]
```

Phrasal edge contains phrases:

```
def phrasal_edge(self):
    return [edge for edge in self.edge() if edge.is_complex()]
```

The edge may contain a unique specifier element that is *licensed* by the head, which is the closest phrasal specifier in the edge that has not been externalized.

```
def licensed_specifier(self):
    if self.phrasal_edge():
        licensed_edge = [edge for edge in self.phrasal_edge() if not edge.externalized()]
        if licensed_edge:
            return licensed_edge[0]
```

5.1.2.10 Criterial feature scanning

```
def scan_criterial_features(self):
    set_ = set()
    if self.left_const and not self.left_const.find_me_elsewhere:
        set_ = set_ | self.left_const.scan_criterial_features()
    if self.right_const and not self.right_const.externalized() and not 'T/fin' in self.right_const.head().features:
        set_ = set_ | self.right_const.scan_criterial_features()
    if self.is_primitive():
        set_ |= {feature for feature in self.features if feature[:3] == 'OP:'}
    return set_
```

5.1.3   Basic structure building

5.1.3.1 Cyclic Merge

Simple Merge takes two constituents α, β and yields [α, β], α being the left constituent, β the right constituent.

It is implemented by the class constructor *__init__,* which takes α and β as arguments and return a new constituent.

```
def __init__(self, left_constituent=None, right_constituent=None):
    self.left_const = left_constituent
    self.right_const = right_constituent
    if self.left_const:
        self.left_const.mother = self
    if self.right_const:
        self.right_const.mother = self
    self.mother = None
    self.features = set()
    self.morphology = ''
    self.internal = False
    self.adjunct = False
    self.incorporated = False
    self.find_me_elsewhere = False
    self.identity = ''
    self.rebaptized = False
    self.x = 0
    self.y = 0
    if left_constituent and left_constituent.externalized() and left_constituent.is_primitive():
        self.adjunct = True
        left_constituent.adjunct = False
```

5.1.3.2 Countercyclic Merge-1

Countercyclic Merge-1 (*merge_1(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]. Thys, if we have a phrase structure [X....α....Y], then Merge-1 generates either (a) or (b).

(57)

a.   [X...[$_\gamma$ α β]...Y]

b.   [X...[$_\gamma$ β α]...Y]

Constituent γ then replaces α in the phrase structure, with the phrase structural relations updated accordingly. 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. The fact that Merge-1 dissolves into separate processes is reflected in the code, which therefore contains three separate functions: the first (*local_structure()*) obtains a snapshot of the local structure around α (its mother and position in the left-right axis), the second creates [α β] or [β α] (*asymmetric_merge()*), and the third (*substitute()*) substitutes α with the new constituent [α β] by using local constituent relations recorded by the first operation.

```
def merge_1(self, C, direction=''):
    local_structure = self.local_structure()          # [X...self...Y]
    new_constituent = self.asymmetric_merge(C, direction)  # A = [self H] or [H self]
    new_constituent.substitute(local_structure)        # [X...A...Y]
    return new_constituent.top()


def asymmetric_merge(self, B, direction='right'):
    if direction == 'left':
        new_constituent = PhraseStructure(B, self)
    else:
        new_constituent = PhraseStructure(self, B)
    return new_constituent


def substitute(self, local_structure):
    if local_structure.mother:
        if not local_structure.left:
            local_structure.mother.right_const = self
        else:
            local_structure.mother.left_const = self
        self.mother = local_structure.mother


def local_structure(self):
    local_structure = namedtuple('local_structure', 'mother left')
    local_structure.mother = self.mother
    local_structure.left = self.is_left()
    return local_structure
```

5.1.3.3   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.

```
def remove(self):
    if self.mother:
        mother = self.mother                   # {H, X}
        sister = self.geometrical_sister()     # X
        grandparent = self.mother.mother       # {Y {H, X}}
        sister.mother = sister.mother.mother   # Y
        if mother.is_right():
            grandparent.right_const = sister   # {Y X} (removed H)
        elif mother.is_left():
            grandparent.left_const = sister    # {X Y} (removed H)
        self.mother = None                     # detach H
```

5.1.3.4   Detachment

Detachment refers to a process that cuts part of the phrase structure out of its host structure.

```
def detach(self):
    if self.mother:
        original_mother = self.mother
        self.mother = None
    else:
        original_mother = None
    return original_mother
```

5.1.4   Nonlocal grammatical dependencies

5.1.4.1   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:

(58) 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. 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.

```
def probe(self, feature, G):
    if self.sister():
        # -------------------- minimal search --------------------------
        for node in self.sister():
            if G in node.inside_path().features:
                return True
            if G[:4] == 'TAIL' and G[5:] in node.left_const.scan_criterial_features():
                return True
            if node.intervention(feature):
                break
        # --------------------------------------------------------------
```

```
def inside_path(self):
    if self.is_primitive():
        return self
    if self.is_complex:
        return self.left_const.head()


def intervention(self, feature):
    feature.issubset(self.inside_path().features)
```

5.1.4.2   Tail-head relations

A tail-head relation is triggered by a lexical feature (TAIL: $F_1, \dots, F_N$), 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. 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.

```
def external_tail_head_test(self):
    tail_sets = self.get_tail_sets()
    tests_checked = set()
    for tail_set in tail_sets:
        if self.strong_tail_condition(tail_set):
            tests_checked.add(tail_set)
        if self.weak_tail_condition(tail_set):
            tests_checked.add(tail_set)
    return tests_checked & tail_sets == tail_sets


def internal_tail_head_test(self):
    tail_sets = self.get_tail_sets()
    if tail_sets:
        for tail_set in tail_sets:
            if self.weak_tail_condition(tail_set, 'internal'):
                return True
            else:
                return False
    return True
```

91

```python
def strong_tail_condition(self, tail_set):
    if self.get_max() and \
            self.get_max().mother and \
            self.get_max().mother.head().match_features(tail_set) == 'complete match' and \
            'D' not in self.features:
        return True


def weak_tail_condition(self, tail_set, variation='external'):
    if 'ADV' not in self.features and len(self.feature_vector()) > 1:
        for const in self.feature_vector()[1:]:
            for m in const.get_affix_list():
                test = m.match_features(tail_set)
                if test == 'complete match':
                    return True
                elif test == 'partial match':
                    return False
                elif test == 'negative match':
                    return False
    if variation=='external' and not self.negative_features(tail_set):
        return False    # Strong test: reject (tail set must be checked)
    else:
        return True     # Weak test: accept still (only look for violations)


def get_max(self):
    ps_ = self
    while ps_.mother and ps_.mother.head() == self.head():
        ps_ = ps_.walk_upstream()
    return ps_


def match_features(self, features_to_check):
    positive_features = self.positive_features(features_to_check)
    negative_features = self.negative_features(features_to_check)
    if negative_features & self.features:
        return 'negative match'
    elif positive_features:
        if positive_features & self.features == positive_features:
            return 'complete match'
        elif positive_features & self.features:
            return 'partial match'


def negative_features(self, features_to_check):
    return {feature[1:] for feature in features_to_check if feature[0] == '*'}


def positive_features(self, features_to_check):
    return {feature for feature in features_to_check if feature[0] != '*'}


def get_tail_sets(self):
    return {frozenset((feature[5:].split(','))) for feature in self.head().features if feature[:4] == 'TAIL'}
```

5.2 Transfer (transfer.py)

5.2.1 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 2.5). The operation targets α and follows a predetermined order: head movement reconstruction $\rightarrow$ adjunct movement reconstruction $\rightarrow$ A´/A-movement reconstruction $\rightarrow$ agreement reconstruction (i.e. Merge-1 $\rightarrow$ Move-1 $\rightarrow$ Agree-1).

```
log(log_embedding + '1. Head movement reconstruction:')
ps = head_movement.reconstruct(ps)
log(log_embedding + f'={ps}')

log(log_embedding + '2. Feature processing:')
feature_process.disambiguate(ps)
log(log_embedding + f'={ps}')

log(log_embedding + '3. Extraposition:')
extraposition.reconstruct(ps)
log(log_embedding + f'={ps}')

log(log_embedding + '4. Floater movement reconstruction:')
ps = floater_movement.reconstruct(ps)
log(log_embedding + f'={ps}')

log(log_embedding + '5. Phrasal movement reconstruction:')
phrasal_movement.reconstruct(ps)
log(log_embedding + f'={ps}')

log(log_embedding + '6. Agreement reconstruction:')
agreement.reconstruct(ps)
log(log_embedding + f'={ps}')

log(log_embedding + '7. Last resort extraposition:')
extraposition.last_resort_reconstruct(ps)

return ps
```

5.2.2 Head movement reconstruction (head_movement.py)

Head movement reconstruction of $[[_\alpha\varnothing, \beta], \gamma]$ can take place in one of two ways:

(59) Head movement reconstruction of $[[_\alpha\varnothing, \beta], \gamma]$ can follow (a.) or (b.)

a. $[[\alpha, \beta], \gamma]$,

b. $[\alpha [_\gamma…\beta…]]$.

Option (a) will be selected if α is complex head with label D, P or A and $[\alpha, \beta]$ satisfies LF-legibility (Section 5.4). In that case no further reconstruction operation will be applied to α. Otherwise option (b) is selected.

```
def reconstruct(self, ps):
    if ps.is_complex():
        ps = self.reconstruct_head_movement(ps)
    elif ps.is_complex_head() and self.left_branch_constituent(ps) and self.LF_legible(ps):
        return self.reconstruct_head_movement(ps)
    return ps


def left_branch_constituent(self, ps):
    return 'D' in ps.features or 'P' in ps.features or 'A' in ps.features


def LF_legible(self, ps):
    return self.reconstruct_head_movement(ps.copy()).LF_legibility_test().all_pass()
```

Head reconstruction (*reconstruct_head_movement()*) travels downward on the right edge of α, targets primitive constituents β at the left that have an affix $\varphi [_\beta\varnothing \varphi]$ and drops the head/affix φ downstream if a suitable position is found.

```
def reconstruct_head_movement(self, phrase_structure):

    # ----------------- minimal search ---------------------------------------#
    for node in phrase_structure:
        if self.detect_complex_head(node):
            complex_head = self.detect_complex_head(node)
            self.controlling_parser_process.number_of_head_Move += 1
            intervention_feature = self.determine_intervention_feature(complex_head)
            self.create_head_chain(complex_head, self.get_affix_out(complex_head),
                    intervention_feature)
    #-------------------------------------------------------------------------#
    return phrase_structure.top()



def detect_complex_head(h):
    if h.is_primitive() and h.has_affix():
        return h
    elif h.left_const and h.left_const.is_primitive() and h.left_const.has_affix():
        return h.left_const
```

Dropping is implemented by fitting the head to the left of each node at the right edge, thus at positions X in

[X α]. 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 (*head_is_selected()*); (ii) it has an EPP feature, has a local

specifier, and can be selected in that position by a higher head (*extra_condition()*).

```
def reconstruction_is_successful(self, affix):
    if not self.head_is_selected(affix):
        return False
    if not self.extra_condition(affix):
        return False
    return True
```

Heads are reconstructed "as soon as a potential position is found." If head reconstruction reaches the bottom,

it will try to reconstruct the head to a position around the bottom node, possibly first reconstructing it if it is

complex. If no legitimate position is still missing, the head will be reconstructed locally to [α [φ XP]] as a

last resort; an unreconstructed head crashes at LF-legibility.

```
def create_head_chain(self, complex_head, affix, intervention_feature):
    if self.no_structure_for_reconstruction(complex_head):
        self.reconstruct_to_sister(complex_head, affix)
    else:
        # -------------- minimal search ---------------------------------------#
        phrase_structure = complex_head.sister()
        for node in phrase_structure:
            if self.causes_intervention(node, intervention_feature, phrase_structure):
                self.last_resort(phrase_structure, affix)
                return
            node.merge_1(affix, 'left')
            if self.reconstruction_is_successful(affix):
                return
            affix.remove()
        # ---------------------------------------------------------------------#
        # Still no solution
        if self.try_manipulate_bottom_node(node, affix):
            return
        else:
            affix.remove()
        self.last_resort(phrase_structure, affix)
```

As shown by the code above, search will also terminate if there is *intervention*.

```
def causes_intervention(node, feature, phrase_structure):
    if node != phrase_structure.minimal_search()[0] and feature in node.sister().features:
        return True
```

Intervention depends on the intervention feature defined by the feature of the complex head.

```
def determine_intervention_feature(self, node):
    if node.has_op_feature():
        return 'D'
    else:
        return '!COMP:*'
```

Last resort is defined as follows.

```
def last_resort(self, phrase_structure, affix):
    phrase_structure.merge_1(affix, 'left')
```

5.2.3   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.

```
def reconstruct(self, ps):
    # --------------------------------- minimal search ---------------------------------------#
    for node in ps.top().minimal_search():
        floater = self.detect_floater(node)
        if floater:
            log(f'\t\t\t\t\tDropping {floater}')
            self.drop_floater(floater, ps)
    # ----------------------------------------------------------------------------------------#
    return ps.top()  # Return top, because it is possible that an adjunct expands the structure
```

A left floater has the following necessary properties:

(60) *A definition of a left 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) it is adjoinable;

(v) it has no criterial features.

```
def detect_left_floater(self, ps):
    if ps.is_complex() and \
            ps.left_const.is_complex() and \
            not ps.left_const.find_me_elsewhere and \
            ps.left_const.head().get_tail_sets() and \
            'adjoinable' in ps.left_const.head().features and \
            '-adjoinable' not in ps.left_const.head().features and \
```

```
        not ps.scan_criterial_features():
        return True
```

Actual floating is triggered by three separate sufficient conditions: (1) the phrase fails the tail-head test; (2) the phrase occurs in finite EPP specifier position; (3) the phrase is a specifier position that is not projected.

```
def detect_floater(self, ps):
    if self.detect_left_floater(ps):
        floater = ps.left_const
        if not floater.head().external_tail_head_test():
            log('\t\t\t\t\t' + floater.illustrate() + ' failed to tail ' + illu(floater.head().get_tail_sets()))
            return floater
        if floater.mother and floater.mother.head().EPP() and 'FIN' in floater.mother.head().features:
            log('\t\t\t\t\t' + floater.illustrate() + ' is in an EPP SPEC position.')
            return floater
        if floater.mother and '-SPEC:*' in floater.mother.head().features and floater == floater.mother.head().local_edge():
            log('\t\t\t\t\t' + floater.illustrate() + ' is in an specifier position that cannot be projected.')
            return floater
```

Condition (2) 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. Right floaters have slightly different properties in the current implementation, because they do not occur in the specifier positions.

```
def detect_right_floater(self, ps):
    if ps.is_complex() and \
            ps.right_const.head().get_tail_sets() and \
            'adjoinable' in ps.right_const.head().features and \
            '-adjoinable' not in ps.right_const.head().features:
        return True
```

Right and left adjunct handling should be unified in the future, but the unification requires extensive empirical adjunct testing. After a floater is detected, it will be *dropped*. Dropping is implemented by first locating the closest finite tense node.

```
def local_tense_edge(self, ps):
    # -------------------------- upstream search --------------------------------------- #
    for node in ps.upstream_search():
        if 'FIN' in node.head().features and node.sister().is_primitive() and 'FIN' not in node.sister().head().features:
            break
    # ----------------------------------------------------------------------------------#
    return 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 of left adjuncts involves three conditions:

(61) *Fitting a floater*

α can be fitted into position P if and only if

(i) tail-head features are satisfied (Section 5.2.4),

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

(iii) P is not a local specifier position.

```python
def conditions_for_left_adjuncts(self, floater, starting_point_head):
    if floater.head().external_tail_head_test() and self.license_to_specifier_position(floater):
        return True

def license_to_specifier_position(self, floater, starting_point_head):
    if not floater.container_head():
        return True
    if floater.container_head() == starting_point_head:
        return False
    if '-SPEC:*' in floater.container_head().features:
        return False
    if not floater.container_head().selector():
        return True
    if '-ARG' not in floater.container_head().selector().features:
        return True
```

Adverbials and PPs will be merged to right, they have to observe only (i).

```python
def conditions_for_right_adjuncts(self, floater):
    if floater.head().external_tail_head_test():
        if self.is_right_adjunct(floater):
            return True
```

The current implementation therefore separates the adjunct conditions for left and right adjuncts. This reflects the fact that they have very different status in the system.

```python
def is_drop_position(self, ps, floater_copy, starting_point_head):
    if self.conditions_for_right_adjuncts(floater_copy):
        return True
    if self.conditions_for_left_adjuncts(floater_copy, starting_point_head):
        return True
```

The floater is promoted into an adjunct, i.e. externalized, once a suitable position is found. This will allow it to be treated correctly by selection, labeling and so on.

```python
def drop_floater(self, floater, ps):
    starting_point_head = floater.container_head()
    floater_copy = floater.copy()
    # ---------------------------------- minimal search ----------------------------------#
    for node in self.local_tense_edge(floater.mother).minimal_search():
        if self.termination_condition(node, floater):
            break
        self.merge_floater(node, floater_copy)
        if self.is_drop_position(node, floater_copy, starting_point_head):
            if not floater.adjunct:
                self.adjunct_constructor.create_adjunct(floater)
            dropped_floater = floater.copy_from_memory_buffer(self.babtize())
            self.merge_floater(node, dropped_floater)
            self.controlling_parser_process.number_of_phrasal_Move += 1
            floater_copy.remove()
            log(f'\t\t\t\t\t = {ps}')
            return
```

```
        floater_copy.remove()
    # --------------------------------------------------------------------------------#
```

5.2.4 External tail-head test

External tail-head is defined in the following way.

(62) 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.

5.2.5 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:

(63) *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

   (*push*);

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

   memory buffer (*pull*);

   (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

   (*push*).

The phrase structure is explored, one head at a time, checking all three conditions for each head. The memory buffer is implemented by the controlling parser process.

```
def reconstruct(self, ps):
    self.controlling_parser_process.syntactic_working_memory = []
    # ---------------------------- minimal search ------------------#
    for node in ps.minimal_search():
        if self.visible_head(node):
            self.pull(self.visible_head(node))
            self.push(self.visible_head(node))
    # --------------------------------------------------------------#
```

Option (i): fill in the SPEC position (push). 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 (pull). 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 βP has no criterial features, it P will undergo A-reconstruction (local successive-cyclicity, Section 5.2.6); 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.

```
def scan_criterial_features(self):
    set_ = set()
    if self.left_const and not self.left_const.find_me_elsewhere:
        set_ = set_ | self.left_const.scan_criterial_features()
    if self.right_const and not self.right_const.externalized() and not 'T/fin' in self.right_const.head().features:
        set_ = set_ | self.right_const.scan_criterial_features()
    if self.is_primitive():
        set_ |= {feature for feature in self.features if feature[:3] == 'OP:'}
    return set_
```

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.

```
[. . .function pull. . .]
if self.must_have_head(spec):
        new_h = self.engineer_head_from_specifier(h, criterial_features)
        iterator.merge_1(new_h, 'left')
    iterator = iterator.mother # Prevents looping over the same Spec element
    if new_h.get_tail_sets():
        self.adjunct_constructor.create_adjunct(new_h)


def engineer_head_from_specifier(self, h, criterial_features):
    new_h = self.lexical_access.PhraseStructure()
    new_h.features |= self.get_features_for_criterial_head(h, criterial_features)
    if 'FIN' in h.features:
        new_h.features |= {'C', 'PF:C'}
    return new_h
```

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 5.2.7).

5.2.6   A-reconstruction

A-reconstruction is an operation in which a DP makes a local spec-to-spec movement, i.e. $[DP_1 [\alpha \_\_\_1 \beta]]$.

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.

```
def A_reconstruct(self, spec, iterator):
    if self.candidate_for_A_reconstruction(spec):
        iterator = self.reconstruct_inside_next_projection(spec, iterator)
    return iterator


def reconstruct_inside_next_projection(self, spec, iterator):
    local_head = spec.sister().head()
    if local_head.is_right():
        # [Y X] => [Y [X Y]]
        local_head.merge_1(spec.copy_from_memory_buffer(self.controlling_parser_process.babtize()), 'right')
        return iterator.mother
    if local_head.is_left():
        if local_head.sister():
            # [Y [X ZP]] => [Y [X [Y YP]]]
            local_head.sister().merge_1(spec.copy_from_memory_buffer(self.controlling_parser_process.babtize()), 'left')
        else:
            # [Y [X <ZP>]] => [Y [X Y] <ZP>]
            local_head.sister().merge_1(spec.copy_from_memory_buffer(self.controlling_parser_process.babtize()), 'right')
        return iterator
```

5.2.7   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.

```
def last_resort_reconstruct(self, ps):
    if self.preconditions_for_extraposition(ps):
        log(f'\t\t\t\tLast resort extraposition will be tried on {ps.top()}.')
        # --------------------------- upstream search ------------------------------------------#
        for node in ps.upstream_search():
            if self.possible_extraposition_target(node):
                self.adjunct_constructor.create_adjunct(node)
                if not node.top().LF_legibility_test().all_pass():
                    log(f'\t\t\t\tThe structure is still uninterpretable.')
        # ----------------------------------------------------------------------------------#
        log(f'\t\t\t\tNo suitable node for extraposition found.')


def preconditions_for_extraposition(self, ps):
    return ps.top().contains_feature('FIN') or 'D' in ps.top().features and not ps.top().LF_legibility_test().all_pass()
```

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].[8]

```
def possible_extraposition_target(self, node):
    if node.left_const.is_primitive() and node.left_const.is_adjoinable() and node.sister():
        if node.sister().is_complex():
            return True
        if node.sister().is_primitive():
            if node.left_const.features & node.sister().complements_not_licensed():
                return True
            if node.sister().get_mandatory_comps() and not (node.left_const.features & node.sister().get_mandatory_comps()):
                return True
```

HP is targeted for possible extraposition operation and HP will be promoted into adjunct (Section 5.2.8). 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:

(64) *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

---

[8] 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.

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 (65).

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

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

5.2.8   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 transfer. 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 current version contains two slightly different versions of the function that creates adjuncts with surrounding structure depending on whether the adjunct is at the correct or incorrect position at the time of adjunct promotion, as determined by the external tail head test.

```python
# Definition for creation of adjuncts
def create_adjunct(self, ps):
    if ps.head().is_adjoinable():
        if ps.is_complex():
            self.make_adjunct(ps)
        if ps.is_primitive():
            if self.adjunct_in_correct_position(ps):
                self.make_adjunct_with_surrounding_structure(ps)
```

```
        else:
            self.make_adjunct_with_surrounding_structure_(ps)
```

The question of whether a specifier must be included is represented in function *eat_specifier*. In essence, the

head must have an edge position which licenses the specifier.

```
def eat_specifier_(self, ps):
    if ps.head().edge() and \
        not '-SPEC:*' in ps.head().features and \
        not (set(ps.head().specifiers_not_licensed()) & set(ps.edge()[0].head().features)) and \
        not ps.edge()[0].is_primitive() and \
        '-ARG' not in ps.head().features and \
        ps.head().mother.mother:
        return True
```

These complications have arisen from empirical work, but they show that we are missing something. What

that missing component is must be determined by systematic empirical inquiry into the properties of adjuncts,

not by trying to "clean up the code." Adjunct promotion checks that the externalized phrase structure is not

the highest node (leaving it without a host), it adds tail features if they are missing and finally transfers the

adjunct to LF.

```
def make_adjunct(self, ps):
    if ps == ps.top():
        return False
    ps.adjunct = True
    self.add_tail_features_if_missing(ps)
    self.transfer_adjunct(ps)
```

5.3   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 requires valuation. That triggers Agree-

1.

```
def reconstruct(self, ps):
    # -------------------------- minimal search ------------------------------- #
    for node in ps.minimal_search():
        if node.left_primitive() and 'VAL' in node.left_const.features:
            self.Agree_1(node.left_const)
    # -------------------------------------------------------------------------#
```

Such heads H attempt to value φ-features. Valuation of φ-features consists of two steps: acquisition of phi-

feature from within the sister of H and, if unvalued features remain, acquisition from the edge.

```
def Agree_1(self, head):

    goal, phi_features = self.Agree_1_from_sister(head.sister())
    for phi in phi_features:
        self.value(head, goal, phi)

    if not head.is_unvalued():
```

104

```
            return

        goal, phi_features = self.Agree_1_from_edge(head)
        for phi in phi_features:
            self.value(head, goal, phi)
```

Operation (1) triggers 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 2019c). Functional heads terminate search.

```
def Agree_1_from_sister(self, ps):
    # -------------------------- minimal search --------------------------#
    for node in ps.minimal_search():
        if node.left_complex():
            if node.left_const.head().is_functional():
                break
        if 'D' in node.left_const.head().features:
            return node.left_const.head(), \
                sorted({f for f in node.left_const.head().features
                        if phi(f) and f[:7] != 'PHI:DET' and valued(f)})
    # ----------------------------------------------------------------------#
    return ps, {}
```

Operation (2) targets the first phrase from the edge in a bottom-up order (adjunct or non-adjunct) with label D and obtains φ-features from it. Notice that the definition of 'edge' includes the pro-element, if present, at H itself. Currently the pro-element is added to the edge and is therefore explored last.

```
# Definition of edge (for Agree-1)
def edge_for_Agree(self, h):
    edge_list = h.phrasal_edge()
    if h.extract_pro():
        edge_list.append(h.extract_pro())
    return edge_list
```

Acquired φ-featured are valued (*value()*) 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.

```
def value(self, h, goal, phi):
    if h.is_valued() and self.valuation_blocked(h, phi):
        h.features.add(mark_bad(phi))
    if find_unvalued_target(h, phi):
        h.features = h.features - find_unvalued_target(h, phi)
        h.features.add(phi)
        h.features.add('PHI_CHECKED')


def valuation_blocked(self, h, f):
    valued_input_feature_type = get_type(f)
    heads_phi_set = h.get_phi_set()
    valued_phi_in_h = {phi for phi in heads_phi_set if valued(phi) and get_type(phi) == valued_input_feature_type}
    if valued_phi_in_h:
        type_value_matches = {phi for phi in valued_phi_in_h if phi == f}
        if type_value_matches:
            return False
        else:
            return True
    return False
```

5.4 LF-legibility (LF.py)

5.4.1 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. The whole phrase structure arriving at LF will always be checked.

```
def test(self, ps):
    if ps.is_primitive():
        self.head_integrity_test(ps)
        self.probe_goal_test(ps)
        self.internal_tail_test(ps)
        self.double_spec_filter(ps)
        self.semantic_complement_test(ps)
        self.selection_tests(ps)
        self.criterial_feature_test(ps)
        self.projection_principle(ps)
        self.adjunct_interpretation_test(ps)
        self.bind_variables(ps)
    else:
        if not ps.left_const.find_me_elsewhere:
            self.test(ps.left_const)
        if not ps.right_const.find_me_elsewhere:
            self.test(ps.right_const)
```

5.4.2 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).

5.4.3   Transfer to the conceptual-intentional system

If the LF-structure passes all tests, it will be transferred to the conceptual-intentional system for semantic interpretation (*transfer_to_CI()*). The operation returns a set (*semantic_interpretation*) containing the semantic interpretation that will then be produced to the output files.

5.5   Semantics (semantics.py)

5.5.1   Introduction

Semantic interpretation is implemented in the module semantics.py. It contains functions which read passively the information in its input and then provide a semantic interpretation as an output. The output will end up in the output files produced by the main script. The output of the semantic interpretation will never affect the internal operation of the syntactic component; it is a passive reflex. However, if no semantic interpretation of any kind results, the expression is still tagged as ungrammatical.

5.5.2   LF-recovery

LF-recovery is triggered if an unvalued phi-feature occurs at the LF-interface. The operation (*LF-recovery*) returns a list of possible antecedents for the triggering head which are then provided in the results and log files.

```
def perform_LF_recovery(self, ps):
    unvalued = must_be_valued(ps.get_unvalued_features())
    if unvalued:
        list_of_antecedents = self.LF_recovery(ps, unvalued)
        if list_of_antecedents:
            self.semantic_interpretation.add(self.interpret_antecedent(ps, list_of_antecedents[0]))
        else:
            self.semantic_interpretation.add(f'{ps}(' + self.interpret_no_antecedent(ps, unvalued) + ')')
        self.report_to_log(ps, list_of_antecedents, unvalued)
```

If both number and person features remain unvalued, this triggers standard control engaging in an upstream path and evaluates whether the sisters of nodes reached in this way evaluate as possible antecedents. The operation is blocked by v* head (head with 'sem:external'). A possible antecedent α must satisfy two conditions: α cannot be a copy of an element that has been moved elsewhere and α must check all semantically relevant phi-features of H. Semantically relevant features are (by stipulation) number, person and definiteness. If no antecedent is found, generic interpretation is generated. If only D_ remains unvalued, then the above mechanism comes with three exceptional properties: If search fails, the expression evaluates as ungrammatical; if there is a local specifier that is not a DP, generic interpretation is generated; v* does not block search. Notice

that the upstream search algorithm includes the head itself in order to see its complement as a potential

antecedent.

```python
def LF_recovery(self, head, unvalued_phi):
    self.controlling_parsing_process.consume_resources("LF recovery")
    list_of_antecedents = []
    if 'PHI:NUM:_' in unvalued_phi and 'PHI:PER:_' in unvalued_phi:
        # ---------------------- minimal upstream search ----------------------------------------------#
        for node in [head] + head.upstream_search():
            if self.recovery_termination(node):
                break
            if node.geometrical_sister() and self.is_possible_antecedent(node.geometrical_sister(), head):
                list_of_antecedents.append(node.geometrical_sister())
        # ---------------------------------------------------------------------------------------------#
        return list_of_antecedents

    if 'PHI:DET:_' in unvalued_phi:
        # ---------------- minimal search--------------------------------------------------
        for node in ps.upstream_search():
            if self.special_local_edge_antecedent_rule(node, ps, list_of_antecedents):
                break
            elif node.sister() and self.is_possible_antecedent(node.sister(), ps):
                list_of_antecedents.append(node.sister())
        #-----------------------------------------------------------------------------
        return list_of_antecedents

    if not list_of_antecedents:
        log(f'\t\t\t\tNo antecedent found, LF-object crashes.')
        self.semantic_interpretation_failed = True
        return []


def is_possible_antecedent(self, antecedent, h):
    if antecedent.find_me_elsewhere:
        return False
    unchecked = get_relevant_phi(h)
    for F in h.get_unvalued_features():
        for G in get_relevant_phi(h):
            check(F, G, unchecked)
    if not unchecked:
        return True
```

6 Formalization of the parser

6.1 Linear phase parser (linear_phase_parser.py)

6.1.1 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.

6.1.2 Recursive parsing function (*_first_pass_parse*)

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. If the user wants to explore all solutions, then the algorithm will backtrack to search for further solutions.

```
def complete_processing(self, ps):
    spellout_structure = ps.copy()
    self.preparations(ps)
    if self.surface_condition_violation(ps):
        return
    ps_ = self.transfer_to_lf(ps)
    if self.LF_condition_violation(ps_):
        return
    if self.transfer_to_CI(ps_):
        return
    self.report_solution(ps_, spellout_structure)
```

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 the 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 (66). 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.

(66) pudo-t-i-n-ko-han   (input word)

    fall-cau-past-1sg-Q-hAn $\rightarrow$ (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.

```
for lexical_constituent in self.lexicon.lexical_retrieval(lst[index]):
    m = self.morphology
    lexical_item, lst_branched, inflection = m.morphological_parse(lexical_constituent, lst.copy(), index)
    lexical_item = self.process_inflection(inflection, lexical_item, ps, lst_branched, index)
    self.number_of_items_consumed += 1
    if inflection:
        self._first_pass_parse(self.copy(ps), lst_branched, index + 1)
    else:
        if not ps:
            self._first_pass_parse(lexical_item.copy(), lst_branched, index + 1)
        else:
```

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 **Virhe. Viitteen lähdettä ei löytynyt.**).

```
adjunction_sites = self.ranking(self.filter(ps, lexical_item), lexical_item)
```

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 [β∅, α]. 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.

```
# ------------------------ consider merge solutions ----------------------------------- #
for site in adjunction_sites:
    ps_ = ps.top().copy()
    site_ = self.node_at(ps_, self.get_position_on_geometric_right_edge(site))
    if site_.bottom_affix().internal:
        new_ps = site_.sink(lexical_item)
    else:
        new_ps = self.transfer_to_lf(site_) + lexical_item
    self._first_pass_parse(new_ps, lst_branched, index + 1)
    if self.exit:
        break
# ------------------------------------------------------------------------------------- #
```

Notice that the left branch phase hypothesis is implemented here.

```
new_ps = self.transfer_to_lf(site_) + lexical_item
```

The complex linearly structured head 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, 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 current implementation uses the function *sink* for this purpose.

```
def sink(self, ps):
    bottom_affix = self.get_affix_list()[-1]
    bottom_affix.right_const = ps
    ps.mother = bottom_affix
    bottom_affix.left_const = None
    return self.top()
```

The structure of *_first_pass_parse* is illustrated in Figure 18.
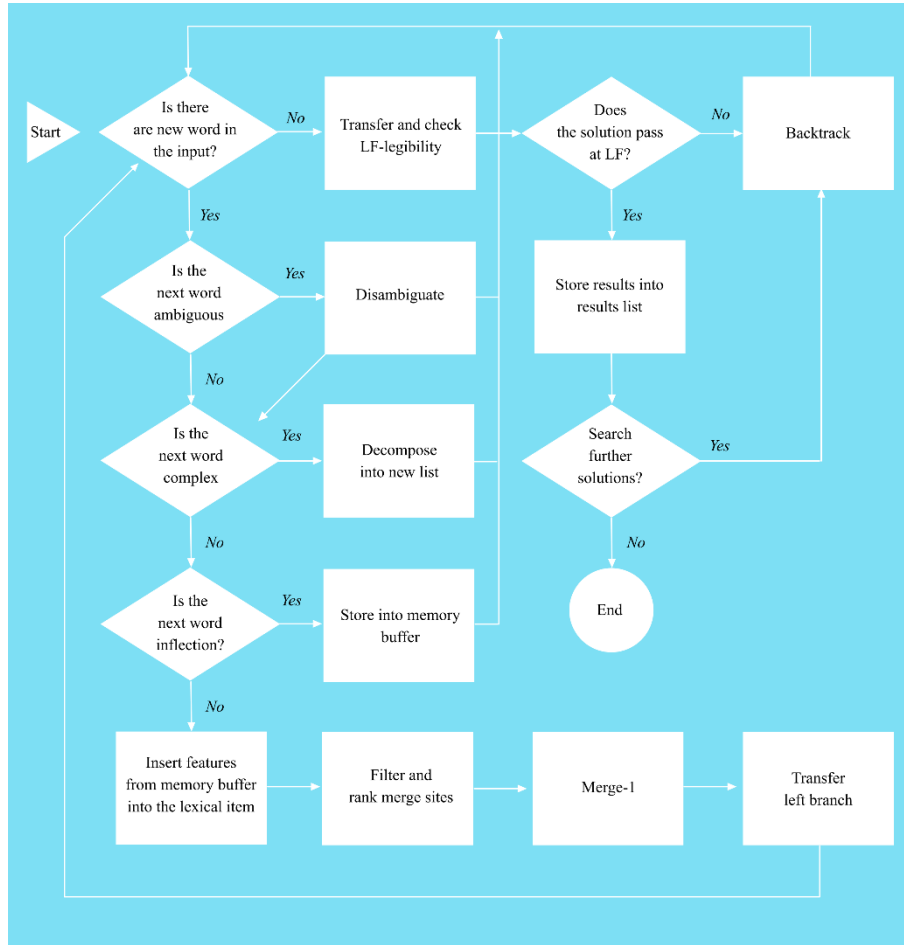
Figure 18. Flowchart of the recursive parsing algorithm *_first_pass_parse()*.

6.2   Psycholinguistic plausibility

6.2.1   General architecture

The parser component obtains a list of all possible attachment sites given some existing partial phrase structure α and an incoming lexical item β. This is list is sent to the module *plausibility_metrics.py* for processing.

```
merge_sites = self.plausibility_metrics.filter_and_rank(ps, terminal_lexical_item)
```

It gets a filtered and ranked list in return, which is used by the parser to explore solutions.

6.2.2 Word internal item?

Suppose α is the existing partial phrase structure and β is the incoming lexical element. If the incoming word β was inside the previous word, then it must be merged to the bottom node of α. The plausibility module examines this fact first and, if it turns out to be true, prunes all other nodes from the list.

```
if self.word_internal(ps, w):
    all_merge_sites = [ps.bottom()]
```

Notice that it is assumed here that the required morphological information is carried from the lexical component into syntax. This is conveyed by the fact that the bottom node of α is marked (by the lexical component) for 'being internal'. The relevant property is defined by the function *word_internal* in this class.

6.2.3 Working memory

If the incoming element β was no inside the previous word in the sensory input, then it can be attached a priori into any node at the right edge of α. The nodes at the right edge of α are first partitioned into two, those which are in the active working memory and those which are not. Only nodes which are in the active working memory are processed further by filtering and ranking. The rest will be simply added to the list in whichever random order they were received:

```
nodes_in_active_working_memory, nodes_not_in_active_working_memory = self.in_active_working_memory(ps)
[...]
all_merge_sites = merge_sites + nodes_not_in_active_working_memory
```

6.2.4 Filtering

Filtering uses several conditions. The operation is implemented by going through all available nodes (i.e. those which are in the active working memory) and by rejecting them if a filtering condition is satisfied. A key property when considering filtering options is that when a parsing branch is closed by filtering, it can never be explored by backtracking; hence we can only close off parsing branches if we can determine based on local information and on local information alone that they will never lead into acceptable structure. The filtering conditions are the following. (1) The bottom node can be rejected if it has the property that it does not allow any complementizers. (2) Node α can be rejected if it constitutes a bad left branch (left branch filter). (3) [α

β] can be rejected if it would break a configuration presupposed in word formation. (4) [α β] can be rejected if it constitutes an impossible sequence.

6.2.5   Ranking (*rank_merge_right_*)

Ranking forms a baseline ranking which it then modifies by using various conditions. The baseline ranking is formed by function *create_baseline_weighting*.

```
self.weighted_site_list = self.create_baseline_weighting([(site, 0) for site in site_list])
```

The weighted site list is a list of tuples (node, weight). Weights are initially formed from small numbers corresponding to the presupposed order, typically from 1 to number of nodes, but they could be anything. This order will be used if no further ranking is applied.

Each (site, weight) pair is then examined and evaluated in the light of *plausibility conditions* which, when they apply, increase of decrease the weight provided for each site in the list. The function returns a list in which the nodes have been ordered in decreasing order on the basis of its weights. Plausibility conditions are stored in a dictionary containing a pointer to the plausibility condition function itself, weight, and logging information. Plausibility condition functions take α, β as input an evaluate whether they are true for this pair; if so, then the weight of α in the ranked list is modified according to the weight provided by the condition itself. The two nested loops implementing ranking are as follows:

```
for site, weight in self.weighted_site_list:
    new_weight = weight
    for key in self.plausibility_conditions:
        if self.plausibility_conditions[key]['condition'](site):
            log(self.plausibility_conditions[key]['log'] + f' for {site}...')
            log('('+str(self.plausibility_conditions[key]['weight'])+') ')
            self.controlling_parser_process.consume_resources('Rank solution')
            new_weight = new_weight + self.plausibility_conditions[key]['weight']
    calculated_weighted_site_list.append((site, new_weight))
sorted_and_calculated_merge_sites = sorted(calculated_weighted_site_list, key=itemgetter(1))
merge_sites = [site for site, priority in sorted_and_calculated_merge_sites]
merge_sites.reverse()
```

In the current version the weight modifiers are ±100. These numbers outperform the small numbers assigned by the baseline weighting. There is nothing in this architecture to force this result. They could compete on equal level if we assumed that the plausibility conditions provide smaller weight modifiers such as ±1. What

the correct architecture is is an empirical matter that must be determined by simulation and psycholinguistic experimentation.

The following plausibility conditions are currently implemented. (1) Positive specifier selection: examines whether [α β] is supported by a position specifier selection feature for α at β. (2) Negative specifier selection: examines whether [α β] is rejected by a negative specifier selection feature for α at β. (3) Break head-complement relation: examines whether [α β] would break an existing head-complement selection. (4) Negative tail-test: examines whether [α β] would violate an internal tail-test at β. (5) Positive head-complement selection: examines if [α β] satisfies a complement selection feature of α for β. (6) Negative head-complement selection: examines if [α β] satisfies a negative complement selection feature of α for β. (7) Negative semantic match: examines [α β] violates a negative semantic feature requirement of α. (8) LF-legibility condition: examines if the left branch α in [α β] does not satisfy LF-legibility. (9) Negative adverbial test: examines if β has tail-features but does not satisfy the external tail-test. (10) Positive adverbial test: examines if β has tail-features and satisfies the external tail-test.

6.3   Morphological and lexical processing (morphology.py)

6.3.1   Introduction

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' exists at the sensory level. Each such element at the PF-interface is 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. A morphologically complex word cannot be merged 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. Decomposition can also be given 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 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).

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 primitive 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.

6.3.2 Formalization

Morphological operations are defined in the module morphology.py. Each lexical item is parsed morphologically *(morphological_parse())*. The operation looks at the current lexical item and detects if it requires decomposition; if it does, then the complex item in the input list is replaced with an inverted list of its constituents. If the first item in the refreshed list is still polymorphemic, the operation is repeated until it is simple and could be merged. The operation also takes care of certain additional special operations (C/op processing, incorporation) that are required for successful morphological parsing.

```
def morphological_parse(self, lexical_item, input_word_list, index):
    lexical_item_ = lexical_item
    while self.is_polymorphemic(lexical_item_):
        lexical_item_ = self.C_op_processing(lexical_item_)
        morpheme_list = self.decompose(lexical_item_.morphology)
        morpheme_list = self.handle_incorporation(lexical_item_, morpheme_list)
        self.refresh_input_list(input_word_list, morpheme_list, index)
        lexical_item_ = self.lexicon.lexical_retrieval(input_word_list[index])[0]
    return lexical_item_, input_word_list, self.get_inflection(lexical_item_)
```

6.4 Resource consumption

The parser keeps a record of the computational resources consumed during the parsing of each input sentence. This allows the researcher to compare its operation to realistic online parsing processes acquired from experiments with native speakers.

The most important quantitative metric is the number of garden path solutions. It refers to the number of final but failed solutions evaluated at the LF-interface before an acceptable solution is found. If the number of 0, an acceptable solution was found immediately during the first pass parse without any backtracking. Number 1 means that the first pass parse failed, but the second solution was accepted, and so on. Notice that it only includes failed solutions after all words have been consumed. In a psycholinguistically plausible theory we should always get 0 expect in those cases in which native speakers too tend to arrive at failed solutions (as in *the horse raced past the barn fell*) at the end of consuming the input. The higher this number (>0) is, the longer it should take native speakers to process the input sentence correctly (i.e. 1 = one failed solution, 2 = two failed solutions, and so on).

The number of various types of computational operations (e.g., Merge, Move, Agree) are also counted. The way they are counted merits a comment. Grammatical operations are counted as "black boxes" in the sense that we ignore all internal operations (e.g., minimal search, application of merge, generation of rejected solutions). The number of head reconstructions, for example, is increased by one if and only if a head is moved from a starting position X into a final position Y; all intermediate positions and rejected solutions are ignored. This therefore quantifies the number of "standard" head reconstruction operations – how many times a head was reconstructed – that have been implemented during the processing of an input sentence. The number of all computational steps required to implement the said black box operation is always some linear function of that metric and is ignored. For example, countercyclic merge operations executed during head reconstruction will not show up in the number of merge operations; they are counted as being "inside" one successful head reconstruction operation. It is important to keep in mind, though, that each transfer operation will potentially increase the number independently of whether the solution was accepted or rejected. For example, when the left branch α is evaluated during [α β], the operations are counted irrespective of whether α is rejected or accepted during the operation.

Counting is stopped after the first solution is found. This is because counting the number of operations consumed during an exhaustive search of solutions is psycholinguistically meaningless. It corresponds to an unnatural "off-line" search for alternative parses for a sentence that has been parsed successfully. This can be easily changed by the user, of course.

Resource counting is implemented by the parser and is recorded into a dictionary with keys referring to the type of operation (e.g., *Merge*, *Move Head*), value to the number of operations before the first solution was found.

```
self.resources = {"Garden Paths": 0,
                  "Merge": 0,
                  "Move Head": 0,
                  "Move Phrase": 0,
                  "A-Move Phrase": 0,
                  "A-bar Move Phrase": 0,
                  "Move Adjunct": 0,
                  "Agree": 0,
                  "Transfer": 0,
                  "Items from input": 0,
                  "Feature Processing": 0,
                  "Extraposition": 0,
                  "Inflection": 0,
                  "Failed Transfer": 0,
                  "LF recovery": 0,
                  "LF test": 0}
```

If you add more entries to this dictionary, they will automatically show up in all resource reports. The value is increased by function *consume_resources(key)* in the parser class. This function is called by procedures that successfully implement the computational operation (as determined by *key*), it increase the value by one unless the first solution has already been found.

```
def consume_resources(self, key):
    if key in self.resources and not self.first_solution_found:
        self.resources[key] += 1
```

Thus, the user can add bookkeeping entries by adding the required key to the dictionary and then adding the line *controlling_parsing_process.consume_resources("key")* into the appropriate place in the code. For example, adding such entries to the phrase structure class would deliver resource consumption data from the lowest level (with a cost in processing speed). Resources are reported both in the results file and in a separate "_resources" file that is formatted so that it can be opened and analyzed easily with external programs, such as MS Excel. Execution time is reported in milliseconds. In Window the accuracy of this metric is ±15ms due to the way the operation system works. A simulation with 160 relatively basic grammatical sentences with the version of the program currently available resulted in 77ms mean processing time varying from <15ms to 265ms for sentences that exhibited no garden paths and 406ms for one sentence that involved 5 garden paths and hence severe difficulties in parsing.

7 Working with empirical materials

7.1 Setup

The empirical data that will be used in the testing a hypothesis or an analysis is first collected into a test corpus and stored into a subfolder in folder *language data working directory*. The name of the test corpus file and the study folder is provided in the configuration file *config.txt* in the root folder. The main script will read and process all sentences from the test corpus file. All output files will be named automatically on the basis of the test corpus file name and generated into the same study folder. The parameters of the parser are provided in an external file *study_config.txt* that must be in the study folder. The study if then launched by the following command to the command prompt that is pointing to the root directory.

```
python lpparse
```

This system corresponds to the most typical use case, in which the researcher runs the script through one test corpus file. Notice that this command will run the module *__main__.py* from the folder *lpparse*. This is done so as to make it possible to direct the program execution into any module by using command prompt arguments. Another use case is if the researcher wants to run several studies at once. This can be done by command

```
python lpparse multi
```

which will run several studies specified in the *multistudy.py* module. The study parameters (study folder, test corpus file) are provided as arguments in the *multistudy.py*, and the user much change them there. Whether the user must run several studies or just one depends on the design of the study. By collecting a multi study into one script makes it possible to replicate the study containing multiple runs.

7.2   Recovering from errors

Running the main script could lead into errors instead of generating an output. This can happen for a variety of reasons. The code could contain a bug, the hypothesis could be formulated in such a way that it is unable to handle some input configuration, or there is an error in one of the input files. Error handling in general is poorly implemented in the present version, and control is returned to the operating system. The user is provided with a console output that determines the source of the error (line in the module causing the error), the type of error, and the recent call structure (how the program called that function). These components are illustrated in Figure 25 below.

Figure 25. An error report printed to the console. The two most important pieces of information are underlined: the location where the error was encountered (module *adjunct_reconstruction.py, line 135*) and the type of the error.

You would then navigate to the indicated location (*adjunct_reconstruction.py, line 135*) and examine the source of the problem. In some rare cases the error is trivial to fix, but in most cases, it is not. This is because the position at which the error is encountered in run-time is not typically the same as its true source. The user will formulate 'hypotheses' concerning the cause and by modifying the code (typically by inserting extra print

or logging comments) exclude all possibilities one by one until left with the true cause. Debugging is nontrivial activity that requires a good command of the programming language.

Some errors only occur when the algorithm processes sentences from the test corpus that have special properties that the researcher has not thought out properly. These errors are important for the theory development. They show that the model can be pushed into an unacceptable conclusion and the theory needs revision or sharpening.

7.3   Observational adequacy

Once the algorithm processes the whole input file, we verify that it reaches the condition of observational adequacy. A grammar that is observationally adequate provides correct grammaticality judgments to all sentences in the test corpus. The simplest way to work with this condition is to use the *grammaticality judgments* file generated by the algorithm. The file is generated each time the main script is run and contains the test sentences together with grammaticality judgments. No other information is provided in this file. The user can then copy this file, rename it, and replace the algorithm output with native speaker judgments. Once done, one can use automatic tools to compare the files. I use Notepad++ comparison plugin. The output is illustrated in Figure 26 for a small ad hoc test corpus and an experimental version of the algorithm. As can be seen, there are numerous errors.
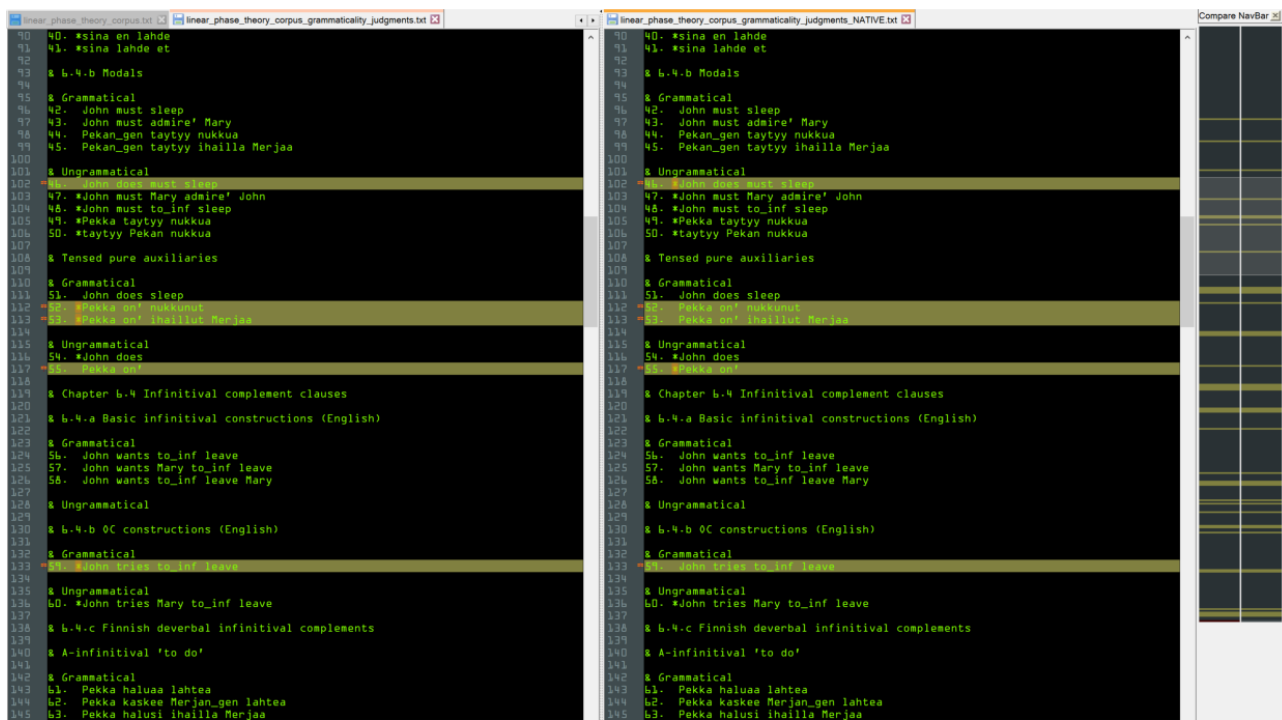
Machine-generated judgments　　Gold standard generated by native speaker

Figure 26. Observational adequacy can be verified quickly by comparing the machine-generated output with a gold standard generated by the user. The narrow panel to the right shows the comparison over the whole file. Here I use the automatic comparison tool available as a plugin for Notepad++.

We will fix any problems, run the algorithm anew, and compare the outputs again until most or all sentences are judged correctly. We have then verified that the analysis is observationally adequate. The simplest way to fix any issues is to take the first sentence judged wrongly, mark it with % in the test corpus, process it alone and then, by examining the output and the derivational log files, find the source of the wrong judgment. Once fixed, we run the whole corpus again.

To examine the cause of wrong judgments, it is almost always necessary to look at the derivational log file. For example, one of the sentences judged wrongly by the experimental algorithm was a canonical interrogative (67) in Finnish. The algorithm judged the sentence ungrammatical.

(67) Ketä　　　Pekka　　　　ihailee?

who.par　Pekka.nom　　admire.3sg

'Who does Pekka admire?'

123

Looking at the derivational log we find that transfer was reconstructing the interrogative pronoun wrongly to SpecvP position, leaving the complement position of 'admire' empty. This, in turn, was because it determined that T had a "wrong complement." Since vP can be selected by T, contrary to what the algorithm was thinking, this indicated that was an error in the part of the theory/code determining whether the complement is right or wrong. This turned out to be the case. Once the code was corrected, the whole test was run anew and the problem was fixed.

7.4　Descriptive adequacy

Once the model reaches observational adequacy, the researcher must verify that the output analyses and semantic interpretations are correct. The algorithm provides several types of output to assist this process. First, the folder */phrase_structure_images* will contain phrase structure images of any solutions generated by the algorithm. The nature of these representations can be controlled by providing several parameters to the main script. A bare bones example of a phrase structure image without any additional information is illustrated in Figure 27. You can add words and their glosses to this image by using the required input parameters.
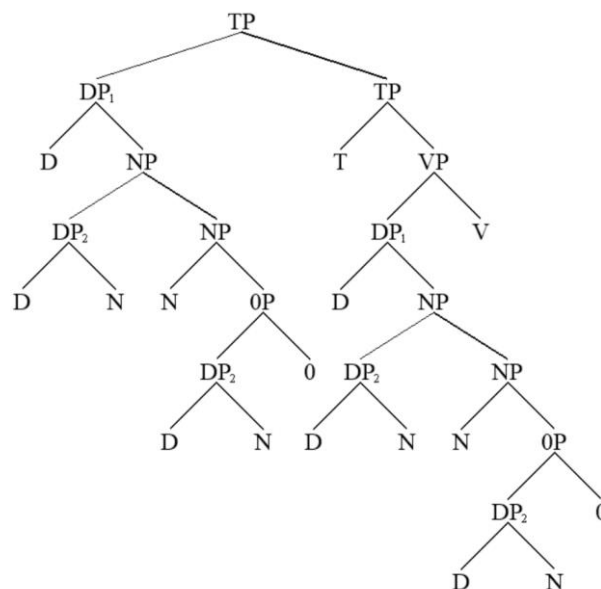

Figure 27. A bare phase structure image.

This output can be used to quickly assess the correctness of the output. The same information plus additional details of the derivation (e.g. semantic interpretation, computational efficiency) are available in the *X_results.txt* file where "X" stands for the name of the test corpus file. Structural analysis are provided in text format in this file. Derivational log is written into *X_log.txt*.

7.5   Performance properties

Performance properties are provided in the form of quantitative performance metrics, the most complete listing available in the resources output. This file contains a list of all sentences processed in the study followed by all quantitative metrics measured at run time. This output can be examined with external programs to obtain summaries of its performance.

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 and Morphosyntax." *Manuscript*.

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

Brattico, Pauli. 2020a. "Case Marking and Language Comprehension: A Perspective from Finnish." *Submitted* x(x):x.

Brattico, Pauli. 2020b. "Computational Linguistics as Natural Science and the Study of Romance Clitics."

Manuscript Submitted.

Brattico, Pauli. 2020c. "Finnish Word Order: Does Comprehension Matter?" *Nordic Journal of Linguistics* x(x):x.

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

Brattico, Pauli. 2020e. "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.