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

(version 9.0)

2019

(Revised May 2021)

Pauli Brattico

IUSS University School for Advanced Studies, Pavia

Abstract

This document describes a computational implementation of a linear phase hypothesis. The model 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 language faculty (UG).

How to cite this document: Brattico, P. (2019). Computational implementation of a linear phase parser. Framework and technical documentation (version 9.0). IUSS, Pavia.

# 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 9.0, which keeps within the framework of the original work but provides improvements, corrections and additions. This document does not, however, substitute for a proper scientific treatment of the topics covered. There are very few references, and the topics are organized into a form of a tutorial that illustrates the core issues by using simplest possible examples. The material is targeted for researchers and computer scientists who need an entry point for understanding the source code. Proper scientific discussion can be found from the published articles. Readers with considerable amount of prior linguistic knowledge on the data and/or theory should consult published sources, which discusses the matter in its proper scientific context. There is also a book length treatment currently under submission that addressed these topics from the cognitive science perspective.

This document is updated as the software component is being developed. There are mismatches between what is described here and what appears in the latest version of the source code. This is because the documentation lags behind and/or has not been done due to the experimental nature of the changes that do not warrant documentation. In addition, some parts of this document are in better condition than others, and there are still gaps awaiting writing. This happens when the material is still being worked on and has not been accepted for publication.

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

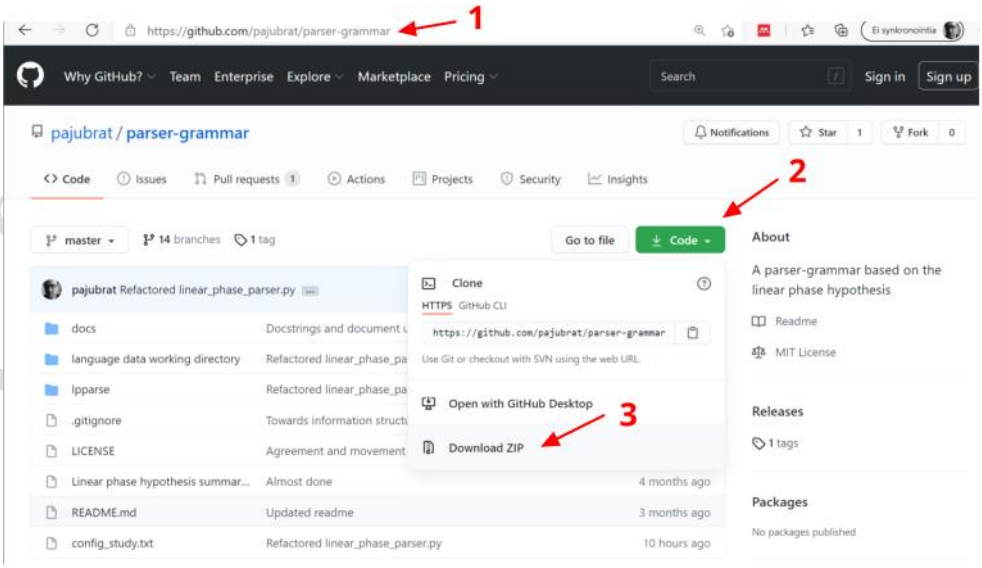
# 2 Installation and basic use

## 2.1 Installation

The linear phase comprehension algorithm is a Python script that processes natural language sentences by using the principles of human language and language understanding. It works by reading test sentences from a test corpus file and analyzing them. The program can be installed on a local computer by cloning it from the source code repository, which can be found at

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

There are several ways for acquiring the program files. The easiest method is by downloading the software package as one ZIP file and extracting it into a directory in the local machine. To do this, navigate to the source repository by using any web browser (1, Figure below), then click the button “Code” (2) and select “Download ZIP” (3).



This will download the software components as one ZIP file. Once you have to ZIP file on the local machine, unpack it into any directory. You should then see the same files and folders as displayed on the above figure on your local machine. The script is ready to use.

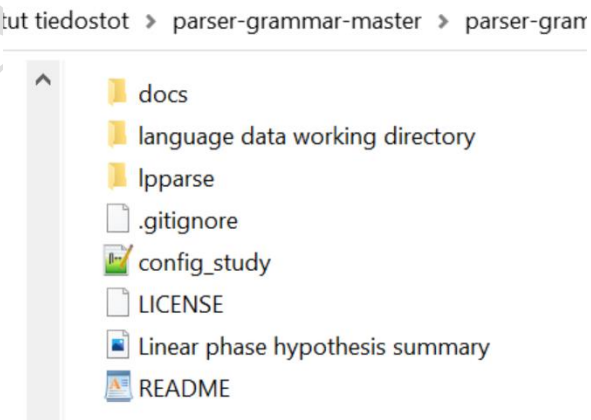
Because the script is written in Python (3x) programming language, you will need Python on your local machine. Python is an interpreter that reads the script (that exists in the form or ordinary text files) and translates it into concrete machine operations. If you do not have Python installed, navigate to the following web address and follow the instructions there:

https://www.python.org/

You can install Python into any directory, it does not have to be and should not be the same directory where you installed the linear phase script. Finally, if you are using Windows, you will have to include the Python installation folder to Windows PATH environmental variable so that Windows can find the Python interpreter when you execute the script. Search for the term "setting windows PATH variables" for instructions on how to do this on your system. The PATH variable must list the folder that contains your Python installation.

## 2.2 Program files and folder structure

At present, the local installation directory (if you download it as a ZIP file) should contain the following files and folders:

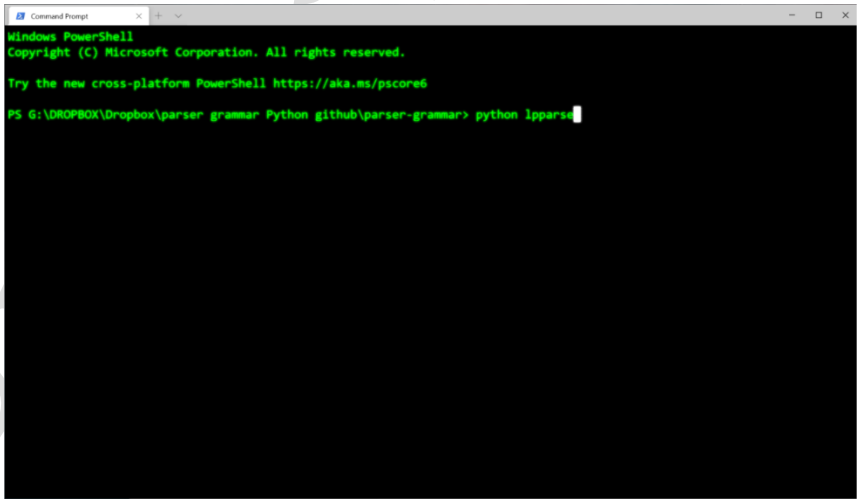



/docs contains documentation (e.g., this document), /language data working directory stores the input and output files associated with any particular study, and /lpparse stores the actual modules containing the program

code. The file *config_study* is a text file that is used to parametrize the operation of the script, in a manner explained below.

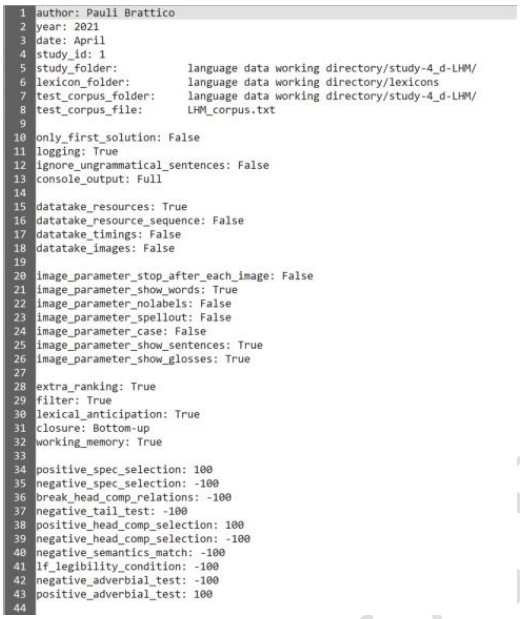
## 2.3 Use

The script is launched by having the Python interpreter to run the program script. In most cases the version you copied or cloned from the code repository comes in a "working configuration," meaning that it should do something meaningful out of the box. Notice, however, that the version you downloaded is always the latest (if the user wants to download older versions, say for replication purposes, then these can be downloaded by first selecting the branch and then performing the instructions above, see below). To launch the script in Windows, start a command prompt program such as Windows PowerShell or the command prompt ("cmd") and then, when you are in the command prompt, navigate to the local installation directory (or launch these programs directly in the installation folder). The script runs by command *python lpparse*, which is written into the command shell as shown below. This asks the Python interpreter (first command *python*) to run the main program module (*__main__.py*) from the folder *lpparse*. I will return to the organization of the program modules below.




The program reads a study configuration file *config_study.txt* from the local installation directory, which contains the parameters used in the simulation trial. This file is a "master control" utility that will tell the script what to do. One of these parameters is the folder and name of the test corpus file that contains the sentences

the script will analyze. The user can open the configuration file by using any text editor (such as Notepad). Here is a screenshot of the file as it exists currently on my local installation directory:



```
1  author: Pauli Brattico
2  year: 2021
3  date: April
4  study_id: 1
5  study_folder:           language data working directory/study-4_d-LHM/
6  lexicon_folder:         language data working directory/lexicons
7  test_corpus_folder:     language data working directory/study-4_d-LHM/
8  test_corpus_file:       LHM_corpus.txt
9
10 only_first_solution: False
11 logging: True
12 ignore_ungrammatical_sentences: False
13 console_output: Full
14
15 datatake_resources: True
16 datatake_resource_sequence: False
17 datatake_timings: False
18 datatake_images: False
19
20 image_parameter_stop_after_each_image: False
21 image_parameter_show_words: True
22 image_parameter_nolabels: False
23 image_parameter_spellout: False
24 image_parameter_case: False
25 image_parameter_show_sentences: True
26 image_parameter_show_glosses: True
27
28 extra_ranking: True
29 filter: True
30 lexical_anticipation: True
31 closure: Bottom-up
32 working_memory: True
33
34 positive_spec_selection: 100
35 negative_spec_selection: -100
36 break_head_comp_relations: -100
37 negative_tail_test: -100
38 positive_head_comp_selection: 100
39 negative_head_comp_selection: -100
40 negative_semantics_match: -100
41 lf_legibility_condition: -100
42 negative_adverbial_test: -100
43 positive_adverbial_test: 100
44
```

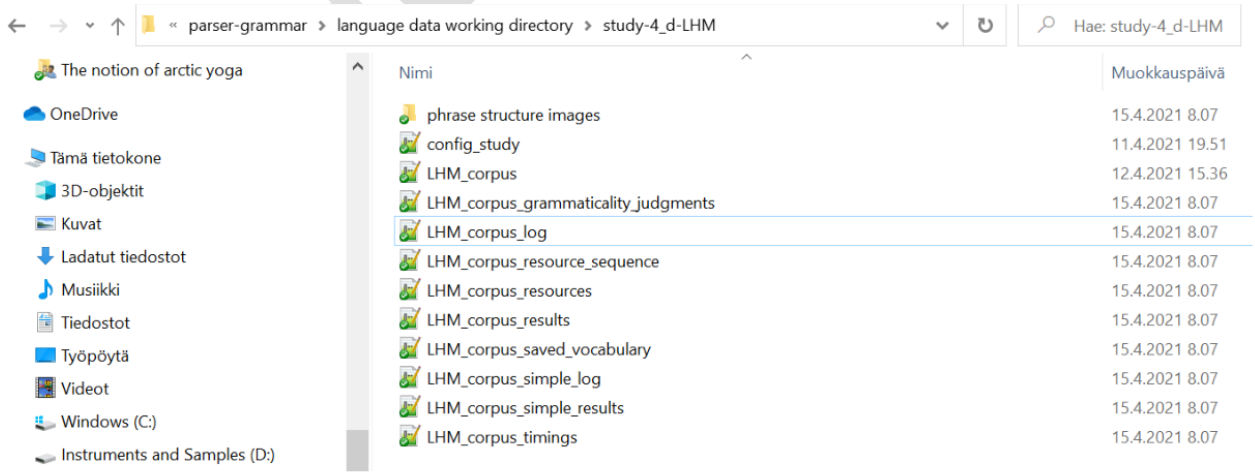
Lines 5-8 in the above screenshot determine the location from which the script will search the test sentences and the lexical files. In this case the script will load a list of test sentences from the folder *language data working directory/study-4_d-LHM* and from file *LHM_corpus.txt* that should be in that folder. The output will be generated into the folder specifier on line 5 (parameter *study_folder*). The rest of the information provides further parametrization, thus whether images are generated, what kind of data will be produced, which heuristic principles are used, and so on.

If I navigate to the test corpus file *LHM_corpus.txt* and open it with a text editor, this is what I see currently:

```
taytyy#[pA]#[hAn] Pekan_gen ihailla Merjaa
taytyy#[foc]#[kO]#[hAn] Pekan_gen ihailla Merjaa
taytyy#[foc]#[pA]#[hAn] Pekan_gen ihailla Merjaa

taytyy#C/op Pekan_gen myyda omaisuuttaan

& Group 2.1.4 Want-to-C

haluaa#[foc] Pekka ihailla Merjaa
haluaa#[hAn] Pekka ihailla Merjaa
haluaa#[kO] Pekka ihailla Merjaa
haluaa#[pA] Pekka ihailla Merjaa
haluaa#[foc]#[hAn] Pekka ihailla Merjaa
haluaa#[foc]#[kO] Pekka ihailla Merjaa
haluaa#[foc]#[pA] Pekka ihailla Merjaa
haluaa#[kO]#[hAn] Pekka ihailla Merjaa
haluaa#[pA]#[hAn] Pekka ihailla Merjaa
haluaa#[foc]#[kO]#[hAn] Pekka ihailla Merjaa
haluaa#[foc]#[pA]#[hAn] Pekka ihailla Merjaa

haluaa#C/op Pekka myyda omaisuuttaan

& Group 2.1.5 Aux-to-C

on'#C/op Pekka ihaillut Merjaa
on'#[foc] Pekka ihaillut Merjaa
on'#[hAn] Pekka ihaillut Merjaa
on'#[kO] Pekka ihaillut Merjaa
on'#[pA] Pekka ihaillut Merjaa
on'#[foc]#[hAn] Pekka ihaillut Merjaa
on'#[foc]#[kO] Pekka ihaillut Merjaa
on'#[foc]#[pA] Pekka ihaillut Merjaa
on'#[kO]#[hAn] Pekka ihaillut Merjaa
on'#[pA]#[hAn] Pekka ihaillut Merjaa
on'#[foc]#[kO]#[hAn] Pekka ihaillut Merjaa
on'#[foc]#[pA]#[hAn] Pekka ihaillut Merjaa

on'#C/op Pekka myynyt omaisuuttaan

& Group 2.1.6 All constructions (2.1.1-2.1.5) with formal C-feature C/fin

ihailee#C/fin Pekka Merjaa
ei#C/fin Pekka ihaile Merjaa
taytyy#C/fin Pekan_gen ihailla Merjaa
haluaa#C/fin Pekka ihailla Merjaa
```

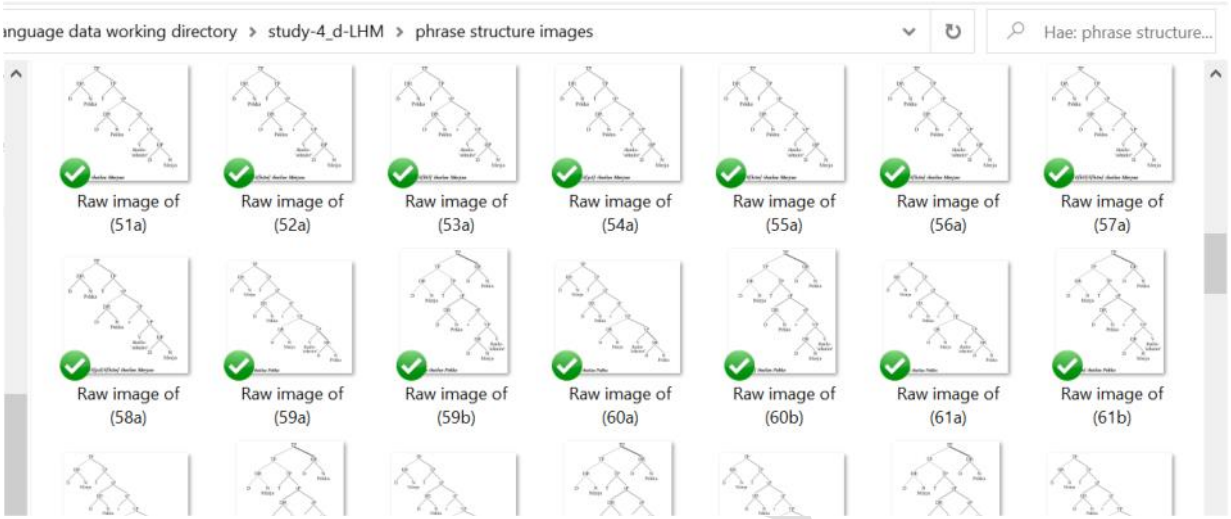
This is a screenshot of the list of sentences that the script will process when it is executed by writing *python lpparse* into the command prompt. If you want to use another file, change the name and folder in the configuration file.

You should see an output in the console as the script processes these sentences. The results are generated into several files inside the study folder, as specified in the configuration file. In this case (as is typical) the study folder is the same as the folder hosting the corpus. This is what I see after running the script:




The first file is the test corpus, followed by several outputs generated by the algorithm. For example, the file *LHM_corpus_grammaticality_judgments.txt* contains a list of the original sentences and the grammaticality

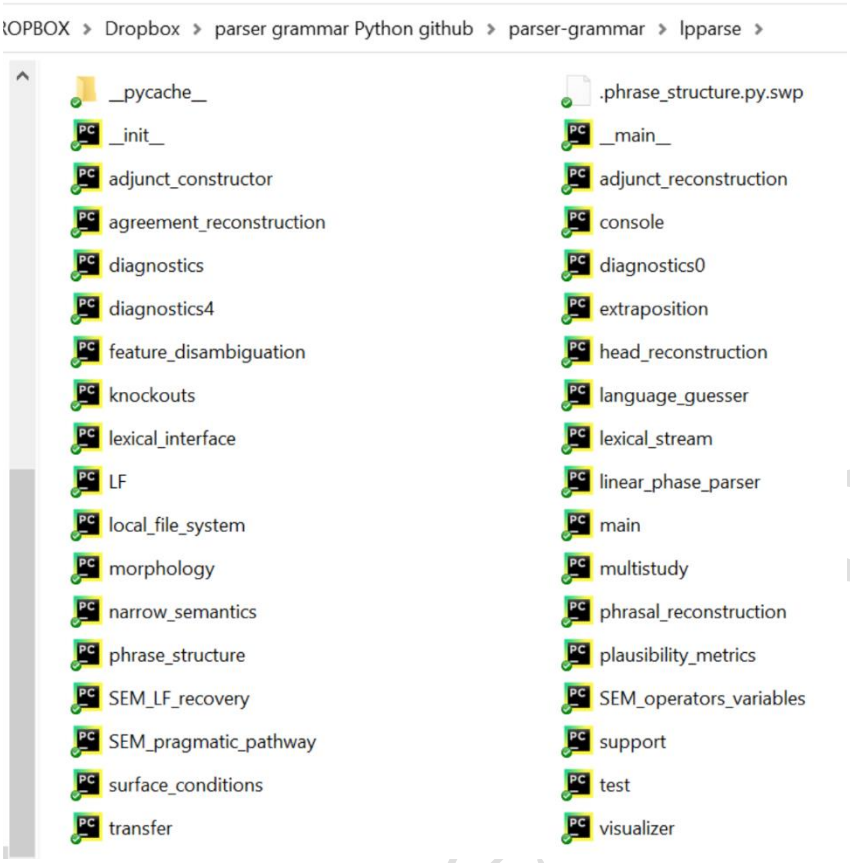
judgments provided by the script. These files can be opened by any text editor. You can also see a folder *phrase structure images* at the top of the directory. If you activated image generation in the configuration file, then this folder will contain phrase structure image for each grammatical input sentence, as provided by the model:




These images, which are raw data produced by the algorithm and replicas of the text output, are useful when the analysis returned by the script is very complex or if the user wants to verify the output quickly. They could conceivable be used in printed research articles. In order to generate these images, the user must install the pyglet library and set image generation on in the configuration file.

## 2.4 Structure of the script

When the user runs the script by typing *python lpparse* inside the installation director, what happens under the hood is that the Python interpreter will run a main script (called *__main__.py*) from the folder /lpparse. This folder contains all the program files and modules which define the theory:

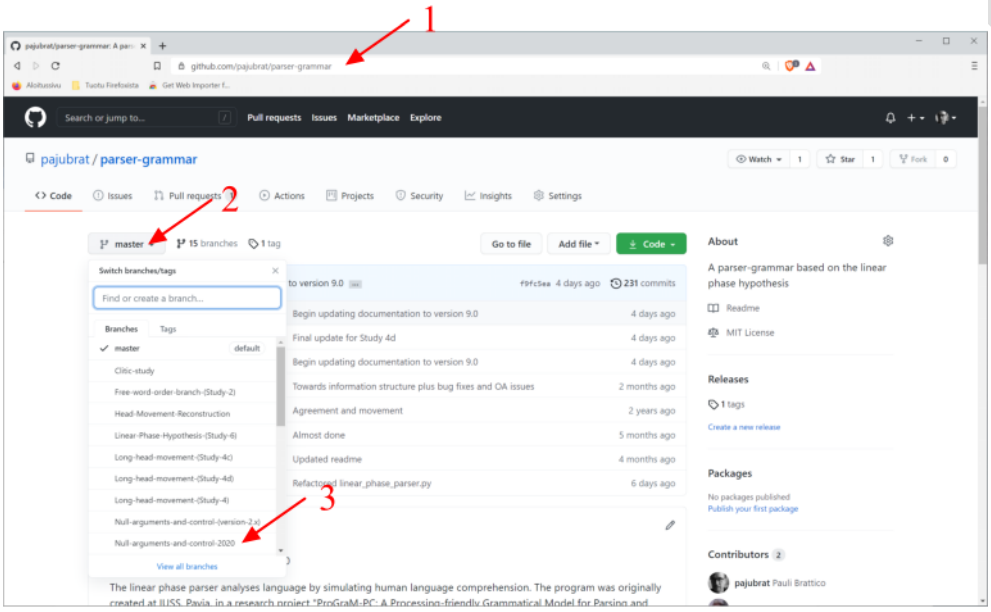



The *__main__.py* module will take control of the execution and call the program code that exists inside the individual files, as required. These files – also called modules – correspond closely to the contents of the empirical theory that they implement. For example, the module *narrow semantics.py* contains operations that correspond to the corresponding empirical component in the theory. These files can be opened and edited by any text editor. Thus, if the researcher wants to find out exactly how some grammatical operation, principle or module operates, this information can be found from these files. For example, the underlying notion of phrase structure is defined in the module *phrase_structure*. If the user changes any of these files and runs the script anew, then the modified script will be run, and the original output files will be overwritten. Python is an interpreted language, meaning that there is no separate executable; each time the user runs the script these text files are consulted.

## 2.5 Replication

When a study is published, the input files plus its source code is stored somewhere. One such place is the source code repository itself. The source code repository (that runs on a program called Git) not only stores

the current version of the script, but also maintains a record of the previous versions. The user can therefore backtrack to a previous point in the development. The development history can also be branched, meaning that it is possible to develop several versions of the software in parallel (and possible merge them later). These parallel branches are currently used to store unambiguous snapshots of the script that were used in connection with published studies. To access them, navigate again to the source code repository (1, Figure below), then click for the tab shown in the figure below (2) and select the branch that is of interest (3).




This should select a snapshot of a development branch that contains a version of the script that was used in a published study. The name of the branch is usually mentioned in the published paper. Use of these branches is not recommended for anything else than replication. They should not be developed further.

# 3 Framework

## 3.1 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, all information required to interpret a sentence in one's native language must be present in the sensory input. The system that performs the mapping from linguistic sensory objects into sets of possible meanings is called a parser, or perhaps more broadly as *language comprehension*. We assume that efficient and successful language comprehension is 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 as outright ungrammatical by native speakers. A realistic 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 its input in terms of grammaticality or some related notion, such as semanticality, marginality or acceptability. These categorizations are studied by eliciting responses from native speakers. Any language comprehension model that captures this mapping correctly will be said to be *observationally adequate*, to follow the terminology from (Chomsky 1964, 1965). Many practical parsers are not observationally adequate. They do not distinguish ungrammatical inputs from grammatical inputs.

Some aspects of the comprehension model are language-specific, others are universal and depend on the biological structure of the human 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 language comprehension 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 theory can be appreciated in the following way. It is possible to design an observationally adequate comprehension theory for Finnish such that it replicates the responses elicited from native speakers, at least over a significant range of expressions, yet the same model 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 model, using different principles and rules, for English. To the extent that the two language-specific models 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 model 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 it 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 in this exact sense.

Suppose we have constructed a theory of language comprehension that is, or can be argued to be, observationally adequate and explanatory. Does it also agree with data obtained from neuro- and psycholinguistic experimentation? Realistic language comprehension involves several features that an observationally adequate explanatory theory need not capture. One such property concerns automatization. Systematic use of language in everyday communication allows the brain to automatize recognition and processing of linguistic stimuli in a way that an observationally adequate explanatory model 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 language comprehension. That being said, the amount of computational resources consumed by the model should be related in some meaningful way to reality. If, for example, the parser engages in astronomical garden pathing when no native speaker exhibits

such inefficiencies, then the model can be said to be insufficient in its ability to mimic real language comprehension. Let us say that if the mode's computational efficiency and performance behavior in general matches with that of real speakers, it 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 linked in the human brain with systems that are responsible for the perception of the same units. What features are shared between the two modes of the language faculty, production and comprehension, is an empirical question we will settle by constructing detailed computational models of both and then by seeing what type of converge is possible.

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. This information processing will be modeled by utilizing processing pathways. 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, are not considered. Because the input consists of phonological words, it is presupposed that word boundaries have been 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. The 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. These semantic interpretations must, furthermore, provide interpretations that agree with how native speakers interpret the input sentences.

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, 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. We could require that the model provides that *John admires Mary* is processed so that John is judged the agent, Mary the patient, and not the other way around. 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, in whichever way they are ultimately represented in the human brain, be predicted. Of course, any given model must ultimately specify how these predictions are generated. I will adopt what I consider to be the standard assumption in the present generative theory and assume that 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 collection of neuronal connections, 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*. 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.

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 lower-level information processing mechanisms that can preprocess such stimuli, perhaps ultimately the acoustic sounds, into a form that 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 access such knowledge. Because we do not allow such annotations, the model must 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 on such grounds. Different theories make different claims regarding 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 few operations. Meaning would then be read off from relatively shallow linguistic representations. An alternative, a "deep" theory of the syntax-semantic interface, is a theory in which the sensory input is subjected to considerable amount of 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 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.

### 3.2 Computational linguistics methodology

Scientific hypotheses must ultimately be justified by deducing the observed facts from the proposed hypothesis. The method, although routinely used in the more advanced sciences, is rarely if ever used in linguistics and therefore requires a comment.

We begin by narrowing down a dataset based on the interests of the particular study. 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 captured by developing a hypothesis. Once the hypothesis is set up, an attempt will be made to calculate its empirical consequences that are compared with the dataset. 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 aim is to provide a mathematical formula that is both sufficient and necessary to deduce the data. Sufficiency is demonstrated by constructing the dataset from the hypothesis, as elucidated above. Necessity is more difficult to show, because it involves an additional concern of showing that the proposed formula is also the simplest (in relation to the largest possible dataset).[2] Although it is hard or impossible to show by relying on completely objective criteria that the hypothesis A is the "simplest" formula possible, if the notion can be made precise at all, we can compare two observationally adequate hypotheses A and B in terms of their simplicity in relation to some dataset. For example, if hypothesis A captures the dataset D by using an explicit

[2] These concerns have played a major role in recent minimalist grammars (Chomsky 1995, 2008). The research literature generated within this framework demonstrates how difficult it is to come up with an agreement on what counts as "simple" or "simplest."

table-lookup model where each input is paired explicitly and by brute force with the correct output while hypothesis B relies on a general mechanism, then hypothesis B will be favored. This is because it generalizes to a much larger dataset; in fact, most linguistic hypotheses generalize over an infinite set of expressions and therefore supersede any finite model. It can be expected that comparisons between competing hypotheses are resolved in the long run by the scientific community as a whole, as has indeed happened, unproblematically, in the more advanced sciences.

The advanced methodology can be applied regardless of the nature of the hypothesis. If the hypothesis involves learning, then the input contains a learning stage and a verification stage. If it is assumed, in addition, that most of the rule of natural language(s) are acquired from experience, then the role of the learning stage will be much more prominent and therefore plays a major role also in the actual demonstration. Specifically, to demonstrate the correctness of such a hypothesis the researcher must show that the target properties of the adult grammar can be acquired by the model after being presented a realistic linguistic input. At the opposite end, if the model involves a considerable innate component, the researcher must show that the same model applies to most or all languages. Both claims require rigorous justification; absent such justification, the claims must be regarded as tentative conjectures.

### 3.3 Overview of the hypothesis

This subsection provides an intuitive and nontechnical introduction to the hypothesis. The hypothesis describes an information processing pipeline that begins from the linguistic sensory input and ends up with meaning. The main components of the model are illustrated in Figure 1.

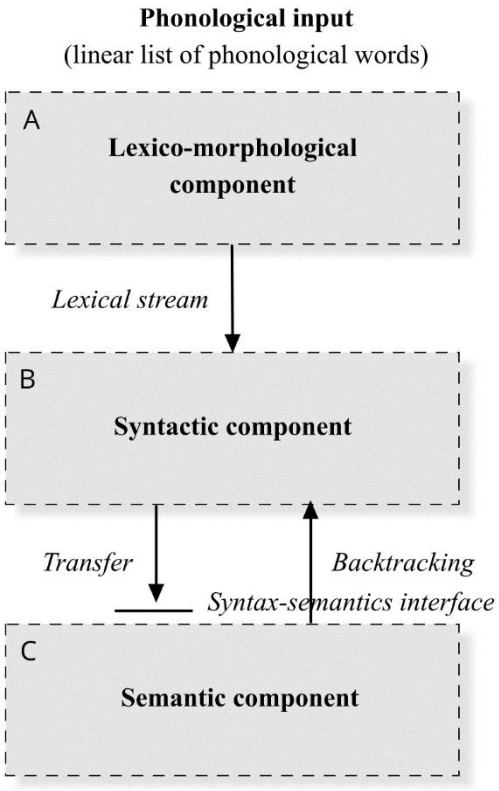



Figure 1. Main components of the model.

The input consists of a linear string of phonological words first processed by a *lexico-morphological component* (A) which retrieves corresponding *lexical items* from the lexicon and forwards them to the syntactic component (B) via a *lexical stream*. If the input word is ambiguous, the lexical items are put into a ranked list and explored in that order. Lexical items are sets of *lexical features*, which constitute the cognitive primitives of the model. The syntactic component (B) attaches incoming lexical items incrementally into a partial phrase structure representation in the current syntactic memory that has been assembled on the basis of the words seen so far. This process is illustrated in Figure 2.

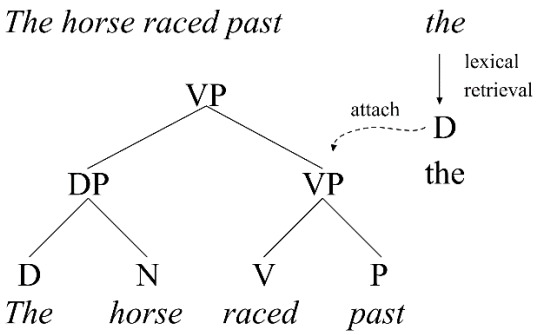



Figure 2. Operation of the syntactic component

The resulting phrase structure representation is called *spellout structure* since it corresponds quite transparently to the linear order in the sensory input, being directly generated form it.[3] Once all words have been consumed from the input, the result will be *transferred* to the *syntax-semantic interface* (C) where the candidate representation is evaluated. If the syntactic interpretation is grammatical and interpretable, the solution will be *accepted* and the input string will be judged grammatical. An accepted structure is forwarded to a semantic component which provides it with detailed semantic interpretation. This corresponds to a process in which the hearer "understands" the input sentence. If the candidate solution is not grammatical and/or cannot be interpreted semantically, it is *rejected* and no semantic interpretation results. In this scenario, the hearer has encountered a difficulty in understanding what the input sentence means. The syntactic component will be notified of the outcome, after which it begins to search alternative solutions by *backtracking*. This operation corresponds to a *reanalysis* of the input, in which the hearer will try to organize the words differently. As stated above, if no acceptable solution emerges, the hearer judges the sentence ungrammatical.

Suppose the input string is *the horse raced past the barn* and the syntactic module provides it with the syntactic representation [$_{VP}$[$_{DP}$ *the horse*] [$_{VP}$ *raced* [$_{PP}$ *past* [$_{DP}$ *the barn*]]]. This input will be accepted at the syntax-semantics interface and interpreted as a declarative clause denoting a specific event containing the horse and

[3] Currently the linearly ordered input string and the spellout structure is mediated by a depth-first left-right linearization algorithm and its (ambiguous) inverse operation.

the barn. If the input string contains an extra word *fell*, however, the first pass parse cannot be interpreted because there is no legitimate position for the last word *fell* (1).

(1) [$_{VP}$ [$_{DP}$ the horse] [$_{VP}$ raced [$_{PP}$ past [$_{DP}$ the barn]]] + *fell* ?

If a solution is rejected, the syntactic component will backtrack (see the arrow "backtrack" in Figure 2) and explore other solutions. The order at which the solutions are explored depends on a number of *heuristic principles* of human language understanding. All attachment solutions that can provide legitimate solutions in principle are explored. Therefore, backtracking allows the model to discover and interpret an acceptable solution (2) with the meaning 'the horse, which raced past the barn, fell'.

(2) [$_{VP}$ [$_{DP}$ the horse [$_{VP}$ (that) raced past the barn]] fell]

The whole information processing pipeline from the phonological input to the syntax-semantic interface is called the *syntactic processing pathway*. It maps input sentences into (sets of) semantic interpretations, with the spellout structures, transfer and the syntax-semantics interface objects serving as intermediate phases.

When the syntactic component assembles a solution for the input, the solution will be *transferred* to the syntax-semantics interface for evaluation and interpretation (see the arrow "Transfer" in Figure 1). While linguistic expressions are language-specific, the system that interprets them is universal. The cognitive capacities of speakers are virtually the same independent of the language(s) they happen to speak. Transfer removes language specific properties from the input so that it can be interpreted and processed further. It detects elements that occur in "wrong" positions where they cannot be interpreted and tries to *reconstruct* them into positions in which they can be interpreted. In Finnish, for example, speakers can reverse the order of the subject and object and produce an inverted OVS sentence that is noncanonical but still grammatical (Finnish is a canonical SVO language). Transfer will reconstruct the object and the subject into their canonical positions where they can be associated at the syntax-semantics interface with universal semantic notions such as agent and patient. In this case the reconstruction is based on the overt morphological case features of the input words

(nominative = subject, partitive = direct object). The process is illustrated in Figure 3, in a highly simplified form.[4]

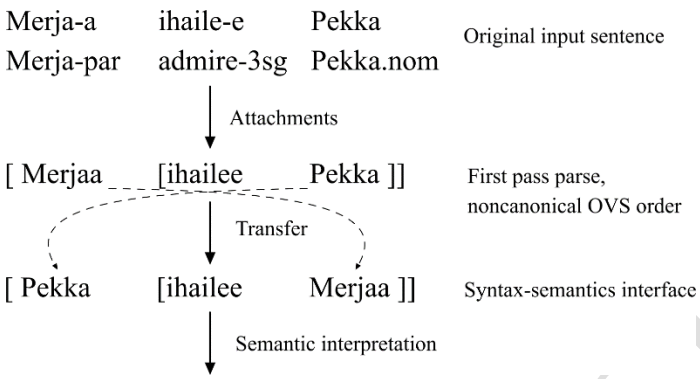



Figure 3. Transfer as an error correction mechanism.

Transfer is a cognitive reflex that is applied to any linguistic representation that it sends to the syntax-semantics interface for interpretation and it always produces the same output, thus there are no alternatives that transfer could explore, recursively or otherwise. We can imagine it as a noise tolerance mechanism performing limited amount of reconstruction when the linguistic elements (words and their pars) appear at noncanonical positions.

Let us consider the nature of the semantic interpretation. At any given moment during a linguistic conversation or communication the hearer maintains a transitory repository of semantic objects called *discourse inventory* that the conversation is "about." Thus, if we talk about a person called John, the discourse inventory will contain a representation of that specific person. The objects maintained in the discourse inventory are language-external objects in the sense that they can be targeted by cognitive processes such as thinking even when no processing occurs in the syntactic pathway. When processing *does* occur in the syntactic pathway, it (like other sensory inputs) will cause changes in the discourse inventory. This corresponds to a situation in which the hearer updates his or her beliefs on the basis of the linguistic input. Semantic interpretation is a process in which the output of the syntactic pathway is converted into changes in the language-external discourse inventory. Each sentence adds, removes or updates elements and their properties in this repository.

[4] Transfer can be said to constitute a "reverse-engineered chain creation algorithm" within the context of modern generative grammar.

This conversion happens inside *narrow semantics*. The relationships between the syntactic pathway, narrow semantics and global cognition are illustrated in Figure 34.

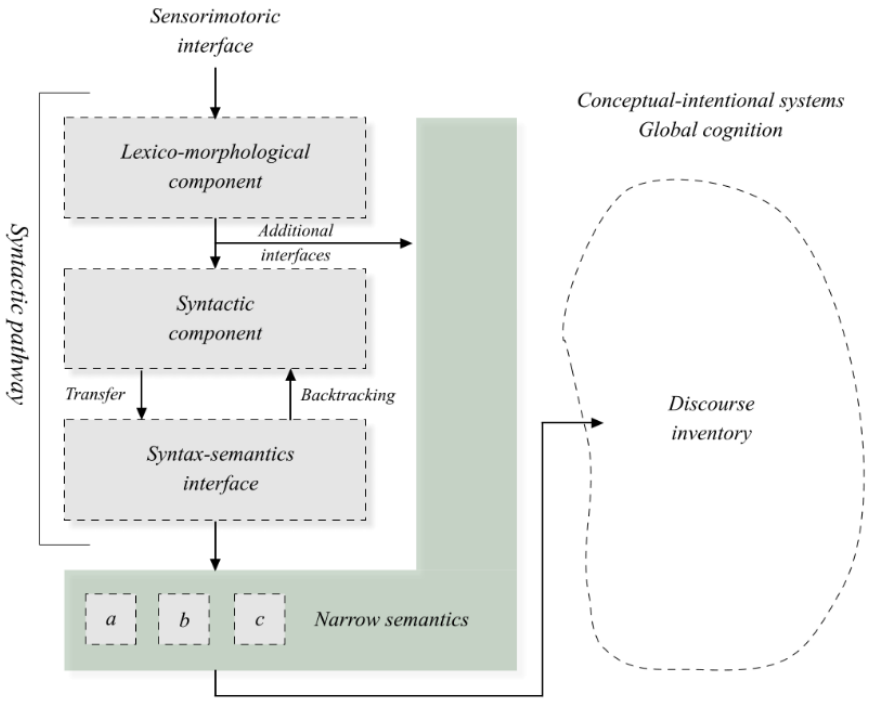



**Figure 34**. The syntactic pathway as embedded inside narrow semantics that mediates all communication between language and global cognition.

# 4 The kernel (comprehension cycle)

## 4.1 Introduction

This section provides minimal specifications that makes it possible to build up the kernel of the model (the comprehension cycle) from scratch. The text is written for somebody working with a concrete algorithm or wants to understand the source code. Empirical matters dealing with the linguistic theory are discussed in published literature and are almost all omitted here. I will begin with the core recursive cycle implemented by the syntactic component (B, Figure 2) and then expand the discussion to other components.

## 4.2 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. One fundamental concern is to recover hierarchical relations between words. Let us assume that while the words are consumed from the input, the core recursive operation in language, Merge (Chomsky 1995, 2005, 2008), arranges them into a hierarchical representation (Phillips 1996, 2003). For example, if the input consists of two words $\alpha * \beta$, Merge yields $[\alpha, \beta]$(3).

(3) 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"? We assume that while the two words in the sensory input are represented as two independent objects, once they are put together in syntax in a manner shown in (3) 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. Operation (3) presupposes that there exists a formally defined notion of phrase structure that is able to represent entities of this type.

Example (3) suggests that the syntactic component combined the phonological words themselves. While this is a possibility, it is not linguistically useful. We assume that the operation illustrated in (3) is mediated by *lexicon*: a storage of linguistic information that is activated on the basis of the original phonological words. The lexicon will map phonological words in the input into *lexical items* which contain features such as lexical categories (noun, verb), inflectional features ('third person singular') and meaning ('John', 'sleeping'). Thus, a word such as *John* is a set of features $\{f_1, \ldots, f_n\}$, its phonological shape and meaning being among them. Features are defined by their formal shape, which are interpreted by various components of the model.

The resulting complex chunk [$\alpha$ $\beta$] is asymmetric and 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, (3) implies that the constituent that arrives first will be the left constituent. Thus, we could think of the left constituent as the "first constituent." Because the configuration is asymmetric, it can be viewed as a linear list that can contain other lists as constituents.

Suppose the next word is *furiously*, which will be merged with (3). There are at least three possible attachment sites, shown in (4), all which correspond to different hierarchical relations between the words.

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

The operation illustrated in (4) differs in a number of ways from what constitutes the standard theory of Merge at the time of present writing. I will label the operation in (4) by Merge-1, 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. 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.

Several factors regulate Merge-1. One concern is that the operation may in principle create a representation that is ungrammatical and/or uninterpretable. Alternative (4)(a) can be ruled out on such grounds: *John furiously* is not an interpretable fragment. Another problem of this alternative is that it is not clear how the adverbial, if it were originally merged inside subject, could have ended up as the last word in the linear input. The default left-to-right depth-first linearization algorithm would produce **John furiously sleeps* from (4)a. Therefore, this alternative can be rejected on the grounds that the result is ungrammatical and not consistent with the word order discovered from the input. If the default linearization algorithm proceeds recursively in a top-down left-right order, then each word must be merged to the right edge of the phrase structure, right edge referring to the top node and any of its right daughter node, granddaughter node, recursively. Under these assumptions a left-to-right model will translate into a grammatical model in which the grammatical structure is expanded at the right edge.

This leaves us either (4)(b) or (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. An incremental parsing process must make decisions concerning an incoming word without knowing what the remaining words are. 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:

(5) *Ranking*

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

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

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

The parser will merge *furiously* into a right-adverbial position (b) and branch off recursively. If this solution does 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. The basic mechanism can be illustrated with the help of a standard garden path sentence such as (6).

(6)

a. The horse raced past the barn.

b. The horse raced past the barn fell.

Reading sentence (b) involves extra effort if compared to (a). 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 (notice that here the garden path is generated due to lexical ambiguity). The correct interpretation is [$_{DP}$ *The horse raced past the barn*] *fell*. 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.

Merge-1 can break constituency relations established in an earlier stage. This can be seen by looking at representations (3) and (4)c, repeated here as (7).

(7) 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 (8).

(8) [John [[sleeps furiously] γ]

One consequence is that upon merging two words as sisters, we cannot know if they will maintain the same or any close structural relationship in the derivation's future. In (8), they don't: future merge operations break up constituency relations established earlier. 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 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 0. It is important to keep in mind that constituency relations established at point *t* do not necessarily hold at a later point *t* + *n*.

### 4.3 Lexical selection features

Consider next a transitive clause such as *John admires Mary* and how it might be derived under the framework as described so far (9).

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

There is evidence that this derivation matches with the correct hierarchical relations between the three words. The verb and the direct object form a constituent that is merged with the subject. If we change the positions of the arguments, the interpretation is the opposite: Mary will be 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. Let us assume that such facts are part of the lexical items: *admire*, but not *sleep*, has a lexical feature !COMP:D which

says that it requires a DP-complement. 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 (10), the test is passed: the label of the selecting item matches with the label of the new arrival.

(10)

| John | admires | Mary |
|---|---|---|
| ↓ | ↓ | ↓ |
| [John | [admires | Mary]] |
| | COMP:D | D |

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. When the parser is trying to sort out an input, it uses lexical features (among other factors) to rank solutions. Remember, however, that these features cannot always be used to filter out solutions, because constituency relations can change in the derivation's future (Section 4.1). But when the phrase structure has been completed and there is no longer any input to be consumed, the same features can be used for filtering purposes. Filtering is performed at the LF interface (discussed later) and will be called the *LF legibility test*.

Let us return to the example with *furiously*. What might be the lexical features that are associated with this item? There are three options in (10): (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 by assuming (again, for the sake of example) that a proper name cannot take an adverbial complement. We are left with two options (11)a-b.

(11)

a. [[$_S$ John [admires Mary]] furiously]

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

Independently of which solution is more plausible (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). 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 at the syntax-semantics interface (LF interface) to verify that the output contains proper specifier-head relations.

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 4.7.1). This constitutes a sufficient (but not necessary) feature for reconstruction; see 4.7.1. 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.

## 4.4 Phases and left branches

Let us consider the derivation of (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. This is because Merge-1 is always to the right edge. All left branches therefore become *phases* in the sense of (Chomsky 2000, 2001). This "left branch phase condition" was argued for by (Brattico

and Chesi 2020) and then adopted into all subsequent models. We can formulate the condition tentatively as (13), but a more rigorous formulation will be given as we proceed.

(13) *Left branch phase condition*

Derive each left branch independently.

All left branches are effectively thrown away from the cognitive working space once they 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.

4.5 Labeling

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

(14) Mary admires John's mother

↓ ↓ ↓ ↓

Mary [admires [ John's mother]]

COMP:D

The verb's 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 rule (15).

(15) *Labeling*

Suppose $\alpha$ is a complex phrase. Then

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

b. if the right constituent 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 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. Conditions (15)c-d mean that labeling ignores right adjuncts (this will be a defining feature of "adjunct").

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 be primitive according to this definition. Consider again the derivation of (3), repeated here as (16).

(16) John + sleeps.

↓ ↓

[John, sleeps]

If *John* is a primitive constituent having no left or right daughters, labeling will categorize [*John sleeps*] as a DP. The left constituent will be the label. This is wrong: (16) is a sentence or verb phrase, not a DP. We assume that *John* is a complex constituent despite of appearing as if it were primitive.[5] Its structure is [D N]. This information comes from the lexicon, where proper names are decomposed into D + N structures. The structure of (16) is therefore (17).

(17) John + sleeps.

↓ ↓

[5] An alternative solution is to reconsider the labeling algorithm. That seems implausible: (15) captures what looks to be a general property of language, thus this alternative would require us to treat (16) and other similar examples as exceptions. There is nothing exceptional or anomalous in (16). The representation should come out as a VP, with the proper name constituting an argument of the VP. Thus, the proper name should not constitute the (primitive) head of the phrase.

D N sleeps

↓ ↓ ↓

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

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 (18).

(18)

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 at the syntax-semantic interface, labels are locked and can no longer undergone change. There are certain situations in which the labels change during transfer.

## 4.6 Adjunct attachment

Adjuncts, such as adverbials and adjunct PPs, present a challenge for any incremental parser. Consider the data in (19).

(19) (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 in essence completely unpredictable. The problem is to define what this type of ‘free attachment’ means.

It is assumed that adjuncts are geometrical constituents of the phrase structure while stored in a parallel syntactic working memory making them invisible for sisterhood, labeling and selection in the primary working memory. This hypothesis is illustrated in Figure 24.

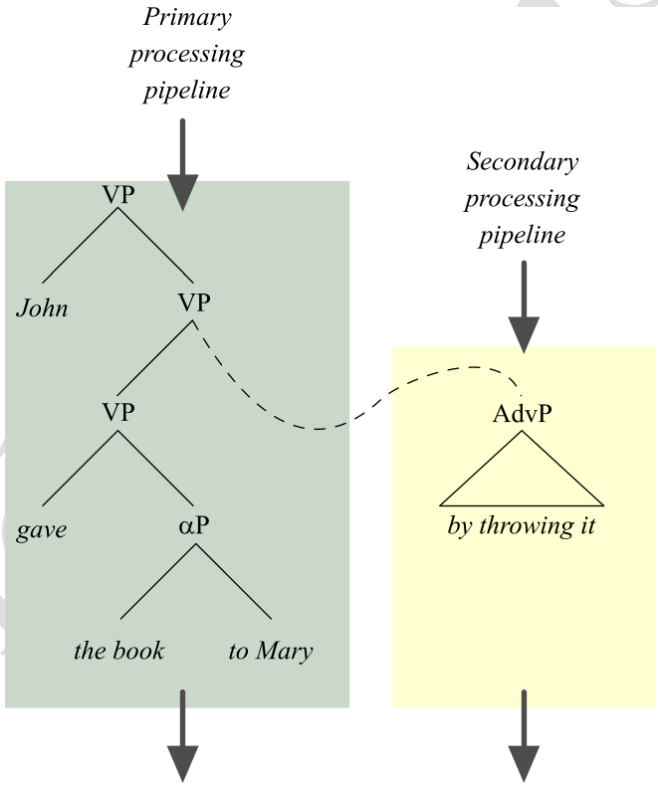



Figure 24. A nontechnical illustration of the way the linear phase comprehension algorithm processes adjuncts. Adjuncts are geometrical constituents that are “pulled out” of the primary working memory and are processed inside separate processing pipeline.

Thus, the labeling algorithm as specified in Section 4.5 ignores adjuncts. The label of (20) 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.

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

Consider (21) next.

(21) 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 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 (21) is $[_{VP}[_{DP}$ *John*$]$ *sleeps*$]$. The fact that adjuncts, like the adverbial *furiously* here, 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. It is assumed that each adverbial (head) is associated with a feature linking it with a feature or set of features 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 (22). Conditions (i-ii) have slightly different content and are applied in different circumstances. This represents an obvious target for future improvement.

*(22) 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 HP 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 in "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 facts differently; the definition provided above has provided the best results so far. 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 4.7.5.

## 4.7 Transfer

### *4.7.1 Introduction*

After a spellout structure phase (left branch, adjunct or the whole sentence) has been composed, it will be transferred to the syntax-semantics interface (LF interface) for evaluation and interpretation. Transfer performs a number of noise tolerance operations which remove most or in some cases all language-specific properties from the input and deliver it in a format understood by the universal semantic system and the universal conceptual-intentional systems responsible for thinking, problem solving and other language-external cognitive operations that are part of the global cognition.

### *4.7.2 A-bar reconstruction*

A phrase or word can occur in a *canonical* or *noncanonical* position. A canonical position in the input could be defined as one that leads the parser-grammar to merge the constituent in that position directly into a position at which it passes all LF interface tests. 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 (23) is a sentence in which the parser reaches a plausible output by merging the elements into the right edge as they are being consumed.

(23) John admires Mary

↓ ↓ ↓

[John [admires Mary]]

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

(24) Ketä Pekka ihaile-e __? (Finnish)

who.par Pekka.nom admire-3sg

'Who does Pekka admire?'

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

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

This solution violates two selection rules: there are two specifiers $DP_1$ and $DP_2$ at the left edge and *admire* lacks a complement. The two violations are related to each other: the element that triggers the double specifier violation at the left edge is the same element that "should be" at the complement position of the verb. We therefore know that the interrogative pronoun causes these problems because it has been "dislocated" to the noncanonical position from its canonical complement position by the speaker. The parser must reverse-engineer the construction in order to create a representation that can be interpreted at the LF interface. These operations, which are called *reconstruction*, take place during transfer.

We could, at least in principle, feed (25) directly to the LF interface component for semantic interpretation. The problem still remains that (25) does not satisfy the complement selection features of the verb *admire*. If we do not control for what is filling the complement position of the verb, the model will not be able to separate grammatical sentences correctly from the ungrammatical ones. It could accept sentences such as **John admires* or even **John admires Mary to leave.* There must therefore occur a process which relates the first element of the clause with the empty position at the end of the clause. Similarly, the main verb is preceded by two argument DPs, yet we cannot allow this to happen generally. A sentence such as **John Mary admires* is

ungrammatical. Reconstruction solves both puzzles by *relating the interrogative pronoun at the beginning of the sentence with its canonical position (26)*.

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

└──────────────────────┘

Why do languages dislocate words and phrases in this manner? It is easy to see that in the example above the dislocated phrase represents the *propositional scope* of the question. The interpretation is one in which the speaker is asking for the identity of the object of admiration ('which x: Pekka admires x'). It holds quite generally that dislocation tends to convey various semantic properties, such as propositional scope, information structure and other similar notions. Let us consider how dislocation represents the propositional scope of the question. 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. Intuitively the first interrogative pronoun represents the fact that the clause is an interrogative and that its scope is the whole clause. To capture this, the algorithm first generates a head between them and copies the interrogative feature into it, as shown in (27).

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

*wh* C(*wh*)

└─────┘

who.par Pekka admires

'Who does Pekka admire?'

The mechanism has two steps. First, we must recognize that C(*wh*) is missing. This 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 new head. The relevant information is obtained from the criterial wh-feature, i.e. from the fact that the fronted 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 is reconstructed to its canonical LF position to satisfy the complement selection for the verb *admire*.

(28) Who does John admire __?

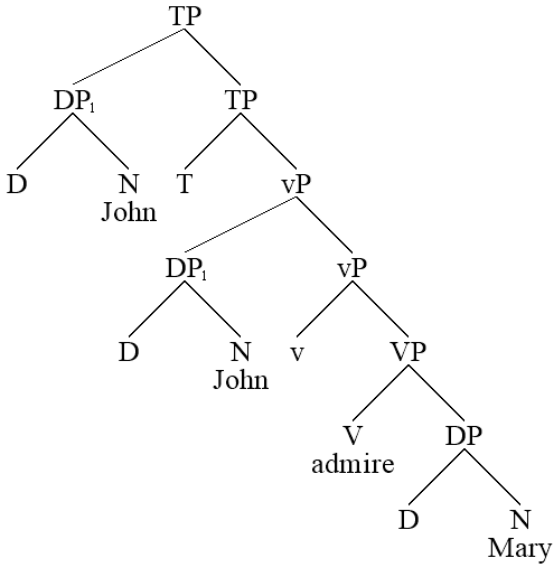
Reconstruction (called Move-1) works by copying the element in the wrong position and reconstructing it. In this case the operation begins from the sister of the targeted element and proceeds downward while ignoring left branches (phases) and all right adjuncts. This operation is called *minimal search*. The element is copied to the first position in which (i) it can be selected, (ii) is not occupied by another legit element, and in which (iii) 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 later operation. If a position is found, it will be copied there; the original element will be tagged so that it will not be targeted second time.

After C(*wh*) has been generated to the structure, all selection features at the left periphery of the clause are potentially checked, yet reconstruction must still occur. One possibility is that the operation is triggered by the missing complement of the main verb. In the present implementation it is assumed that error recovery is triggered at the site of the element that needs reconstruction. Specifically, 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 thematically (in the present formalization, labels select and are selected). Reconstruction is triggered by the presence of this feature (Brattico and Chesi 2020).

### *4.7.3 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 (29).

(29)




*4.7.4 Head reconstruction*

Head reconstruction solves the problem with complex heads that they cannot be interpreted at the LF interface and must be sanitized during transfer. For example, a complex head such as T-v-V contains contradictory lexical features (T having a formal EPP requirement while v needs to check the presence of a thematic agent DP) and thus it is not clear how they should or could be used for the purposes of semantic interpretation. Complex heads are effectively uninterpretable.

Complex heads are generated from phonologically complex words by the lexico-morphological module, which converts phonologically complex heads into lopsided phrase structure objects that do not have the left constituent. This lopsided character means that phrasal syntactic operations do not treat them as phrases (observing the lexical integrity principle), while they can still contain several grammatical heads (primitive lexical items) ordered into a linear sequence. To illustrate, consider (30).

(30) 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 *nukku-a-ko* 'sleep-T/inf-Q' contains elements that are in a 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. The lexical and morphological component 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 *nukku-a-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 (31).

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

C $T_{inf}$ V Pekka thought that he must

The heads are sent in this order to the syntactic component via a lexical stream (Section 8.1) and are collected into one complex head by the syntactic parser. The incoming morphemes are stored into the right constituent of the preceding morpheme which creates a linearly ordered sequence of primitive morphemes. Thus, if syntax receives a word-internal morpheme β, it will be merged to the right edge of the previous morpheme α: [$_{\alpha}$ β]. Notice that by defining the notion of complex constituent as one that has both the left and right constituent, [$_{\alpha}$ β] will come out as a primitive head. The linear sequence C-T/inf-V becomes [$_{C}$ [$_{T/inf}$ V]]. This is what gets merged (32).

(32) [$_{C}$ [$_{T/inf}$ V]] Pekka ajatteli että hänen pitää __.

Pekka thought that he must

Representation (32) is not a legitimate LF-object. The first constituent [$_{C}$ [$_{T/fin}$ V]] cannot be interpreted and the embedded modal verb *pitää* 'must' lacks a complement. Both problems are solved by an operation which reconstructs [$_{T/inf}$ V] from C into the underlying structure. This is done by finding the closest position where T/inf can be selected from above and where it does not violate any 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 $T_{inf}$.

(33) [$_{C}$ __] Pekka ajatteli että hänen [pitää [[$_{T/inf}$ V] ]].

Pekka thought that he.gen must

└──────────────────────────────────────┘

The existence of head movement is an established fact, but it has presented researchers with an enigma: what possible purpose could these manipulations serve? To examine the question form the point of view comprehension, let us consider (34).

(34) 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 fronted head carries the yes/no question particle *-kO* (glossed as Q) that represents the fact that the sentence constitutes an interrogative. In addition, it determines the scope of the question. Both properties are derived by assuming that the fronted head and the Q-feature generate a C(Q) into the structure that will later be interpreted by narrow semantics and its operator-variable module as a scope marker. The head, together with a copy of the Q-feature, are then reconstructed into the canonical position. The operator-variable module, which is part of narrow semantics, will interpret the resulting construction as expressing a yes/no question 'targeting' the predicate:

(35) C(Q) Pekka ajatteli että hänen pitää nukku-a?

C(Q) Pekka thought that he must sleep-A/inf.Q

Scope ←——————————————— Predicate + Yes/No operator

In this case, then, head movement represents (35): it allows the language faculty to target a predicate by a left peripheral Q-feature. While this rationalization applies in some cases, it does not apply in every case. The packing and unpacking of the verb and the A-infinitival suffix is presumably not related to scope. In English, the same construction is expressed by using two separate words *to + sleep*. What seems to be happening is some type of sensorimotoric compression. When a head is packed inside another head, it is not just the head itself that effective disappears from the phrasal syntax; rather, what disappears are both the head and the phrasal positions it projects and licenses. In an agglutinative language like Finnish much of the

syntactic structure is eliminated from the spellout structure in favor of morphologically complex words and therefore must be reconstructed – literally manufactured into existence – by the language comprehension algorithm.

### *4.7.5 Adjunct reconstruction*

Consider the pair of expressions in (36) and their canonical derivations.

(36)

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. 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, 2019a, 2020, 2021). See (Baker 1996; Chomsky 1995: 4.7.3; Jelinek 1984) for similar hypothesis. Case forms are viewed as morphological reflexes of tail-head features. 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 **Virhe. Viitteen lähdettä ei löytynyt.** is satisfied. In this way, the inversed subject and object can find their ways to the canonical LF-positions (37). Notice that because

the grammatical subject *Pekka* is promoted into adjunct, it no longer constitutes the complement; the partitive-marked direct object does.

(37) [⟨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 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.

### *4.7.6 Minimal search*

All reconstruction uses minimal search for locating the reconstruction site. The term "minimal" comes from the fact that the first acceptable position is always selected. The algorithm assumes a phrase structure $\gamma$ as input and moves to $\alpha$ in $\gamma = \{_{\alpha P}\ \alpha\ \beta\}$ unless $\alpha$ is primitive, in which case $\beta$ is targeted.[6] The operation therefore moves downwards on the right edge of the phrase structure and follows selection and labelling. Left and right (adjunct) phrases are dodged. The operation never branches since the algorithm provides a unique solution for any constituent. There is a possible deeper motivation for minimal search. Both left branches and right adjuncts are transferred independently and therefore they are no longer in the current syntactic working memory. It seems, in other words, that minimal search coincides with the contents of the current syntactic working memory at any point in the derivation. If this hypothesis can be maintained, then we could replace the current definition with 'search the contents of the syntactic working memory in top-down order'.

### *4.7.7 Agree-1*

Most languages exhibit an agreement phenomenon illustrated in the example (38).

[6] Notation $\{\alpha\ \beta\}$ refers to $[\alpha\ \beta]$ or $[\beta\ \alpha]$, depending on the actual case.

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

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 in this document) of one element, typically a DP, covary with the same features on another element, typically a verb, as in (38). Not all lexical elements express phi-features, and those which do can be separated into three classes with respect to the type of agreement that they exhibit. Whether a lexical item exhibits agreement 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 can exhibit agreement have feature +VAL. Heads which phi-agree can be divided further into two groups: those which exhibit full phi-agreement with an argument and those which exhibit concord. Whether a lexical item exhibits full phi-agreement with a full argument is determined by feature ±ARG. Negative marking −ARG creates concord, the positive marking +ARG forces the element to get linked with a full argument DP. This linking will be interpreted at LF interface as *predication*. These features leave room for a predicate that is linked with an argument but does not phi-agree. This group creates 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 exhibiting neither 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*) |

An argument DP has interpretable and lexical phi-features 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. These features will be abbreviated as φ. A predicate must be linked with an argument that has phi-features. To

model this asymmetry, a predicate with +ARG will have *unvalued* phi-features, denote by symbols φ_. The value will be provide by the argument with which the predicate is linked with. This is shown in (39).

(39) 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 of a predicate on the basis of an argument with which the predicate is linked with (40). Agree-1 is applied only to heads with +VAL.

(40) 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 Agree-1, the unvalued features disappear from the lexical item and the predicate is no linked with its argument. If no suitable argument is found, unvalued features remain. This scenario will be discussed in the next section.

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 (41) comes from Italian.

(41) adoro Luisa.

admire.1sg Luisa

'I admire Luisa.'

Inflectional phi-features of predicates are extracted from the input and embedded as morphosyntactic features to the corresponding lexical items as shown in (42).

(42) John admire + s Mary.

↓ ↓ ↓

[John [admire Mary]]

{…3sg…}

This means that *admires* will have both unsaturated phi-features and saturated phi-features as it arrives to syntax. Unsaturated features will still require valuation, which triggers Agree-1. The existing valued phi-features, if any, impose two further consequences to the operation. First, Agree-1 must check that if the head has valued phi-features, no phi-feature conflict arises. 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 (43).

(43) adoro Luisa.

admire.1sg Luisa

admire.pro Luisa

'I admire Luisa.'

Agree-1 is limited to local domain. 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.

### *4.7.8 Ordering of operations*

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 (44).

(44) *Ordering of operations during transfer*

Merge-1(α, β) →

Transfer α:

Reconstruct heads →

Reconstruct adjuncts →

Reconstruct A/A'-movement →

Agree-1.

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.

## 4.8 Lexicon and morphology

Most phonological words enter the system a polymorphemic units that might be further associated with inflectional features. The morphological component decomposes phonological words into these components. The lexicon 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 (45). Notice the reversed order.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| (45) | 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 'sleep#T/fin#Q', where each morpheme *sleep*-, T/fin and Q is matched with a further lexical item. 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, converted into features and then inserted inside adjacent lexical items.

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*.

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 rule states that the feature v implies 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, the language-specific lexicon wins.

## 4.9 Narrow semantics

### *4.9.1 Introduction*

Semantics is the study of meaning. There are as many variations of what semantics is as there are variations of the notion of "meaning." In this study we construct the notion of "meaning" in the following way. We assume that linguistic conversation and/or communication projects a set of semantic objects that represents the things that the ongoing conversation or communication is about. It may therefore hold things like persons, actions, thoughts or propositions, as represented by the hearer and the speaker. This temporary semantic repository is called *discourse inventory*. The discourse inventory can be accessed by global cognition, thus operations like thinking, decision making, planning, problem solving and others. Linguistic expressions are

utilized to introduce, remove and update entities in the discourse inventory. We assume that these update operations give linguistic expressions their meaning, in a narrow sense relevant to the present study.

The hypothesis that linguistic expressions provide a vehicle for updating the contents of the discourse inventory requires that there exists some mechanism that translates linguistic signals coming from the syntactic pathway into changes inside the discourse inventory. This mechanism, or rather collection of mechanisms, is called *narrow semantics*. It can be viewed as a "shell" or gateway that encapsulates the syntactic pathway and mediates communication in and out. Cognitive systems that are outside of narrow semantics belong to global cognition. Narrow semantics is implemented by special-purpose functions or modules which interpret and process features arriving through the syntax-semantics interface and then by using the results to populate and update the discourse inventory. The overall structure is illustrated in Figure 34.

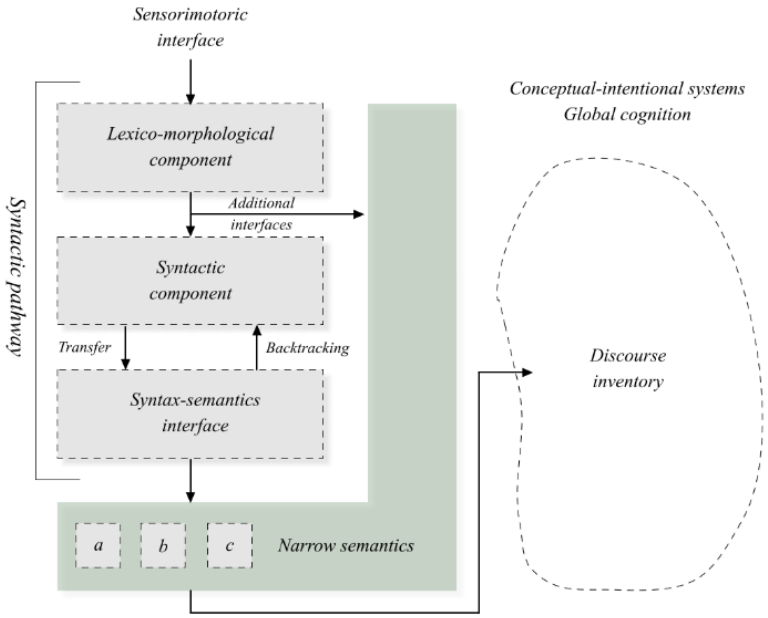



**Figure 34**. The overall relationships between various components of the model.

Different grammatical features are interpreted by qualitatively different semantic systems. Information structure (notions such as topic and focus) is created by different processing pathway than the system that interprets quantifier scope. Narrow semantics therefore constitutes a router at which the processing of different features is distributed to different language-external systems. In order to accomplish this, the system needs to

know which features are feed into which systems. This is currently determined on the basis of the formal shape of the features. Operator-variable features (Section 4.9.3), for example, are recognized as such on the basis of their surface form [OP:]. Thus, narrow semantics responds to lexical features on the basis of their shape and sends them forward to various subsystems that can interpreted them.

### *4.9.2 Semantic wiring and the projection of the discourse inventory*

The discourse inventory maintains a transient storage of semantic objects that are part of the present discourse. For example, if the speaker mentions a person by using a proper name such as *John*, the corresponding semantic object 'John' is created into the discourse inventory and a link is established between the expression and the semantic object. The link is maintained by a *referential index* feature [IDX:N] of the referring expression that points to a unique object inside the discourse inventory. When a semantic object is created, the system will use linguistic features present in the expression to infer its *semantic type*. A proper name such as John will be linked with a semantic object whose type of §Thing, referring to a spatiotemporal "thing" that is realized as an uninterrupted trajectory in space and time.[7] This information is stored into the discourse inventory together with the object itself and possible other information that is relevant for linguistic processing and global cognition. It is assumed that these objects and the corresponding semantic types form the substance of human conceptual thinking. The discourse inventory is a transient storage of such elements that are maintained during "one conversation."

### *4.9.3 Interpretation of operator-variable constructions*

Operator-variable features all have the distinctive shape [OP:F] where F is any string, which causes narrow semantics to send them into the operator-variable module. The operator-variable module interprets all operator features by matching them with a propositional scope marker that must be marked with [OP:F][FIN]. How the scope marker is generated depends on the language: in Finnish it will be generated during A-bar reconstruction, as explained in Section 4.7.1. This interpretation also involves the creation of the corresponding semantic

[7] Cognitive categories like §Thing are psychological categories, not "metaphysical" building blocks.

objects (Operator, Proposition) into the discourse inventory, so that other language external systems have access to them. Further interpretation is determined by the nature of F.

### *4.9.4 Argument structure*

Argument structure refers to the structure of thematic arguments and their predicates at their canonical LF interface 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, only 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 SpecvP, SpecVP and CompVP, whereas an intransitive two, SpecVP and CompVP. 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 (46). 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.

(46)

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.

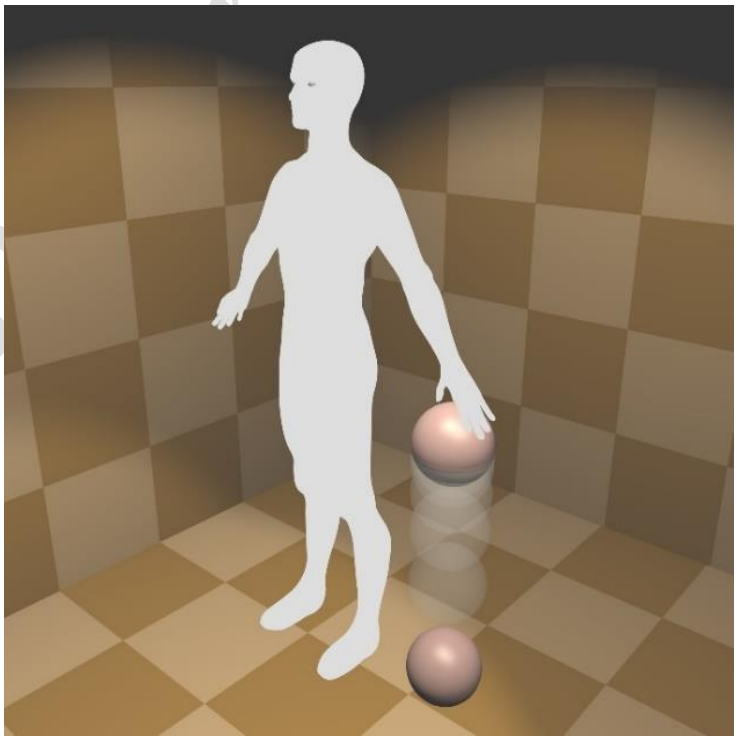


Figure 8. A non-discursive, perceptual or imagined representation of the meaning of the sentence *John dropped the ball*.

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*.

### *4.9.5 Antecedents and control*

The assumptions specified in the section elucidating Agree-1 (4.7.7) 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. 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. LF-recovery is activated inside narrow semantics which calls for the module performing the recovery itself. 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:

(47) John wants to leave$_{\varphi_ = \text{John}}$

‘John$_1$ wants: John$_1$ to leave.’

The upward path 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’ (48).

(48) 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 (49).

| | | | | |
|---|---|---|---|---|
| (49) 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.

### *4.9.6 The pragmatic pathway*

To be written (paper is currently under revision)

# 5 Performance

## 5.1 Human and engineering parsers

The linear phase theory is a 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.

## 5.2 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.

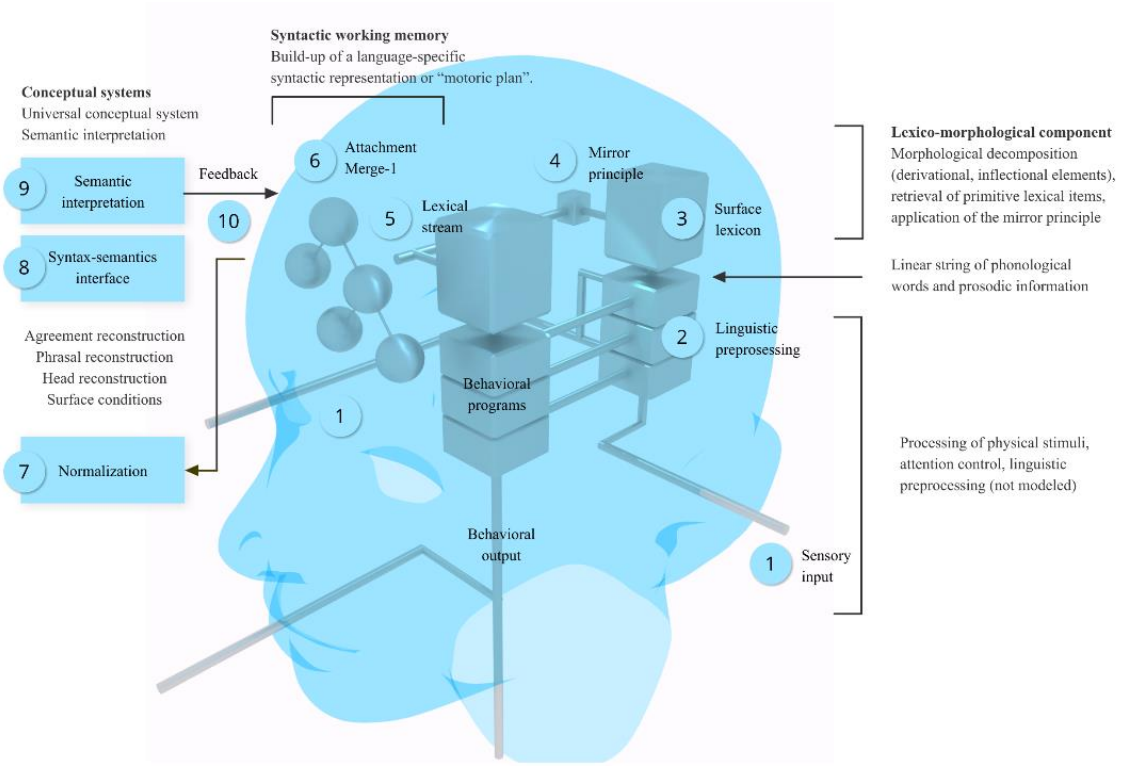



Figure 32. Components of the model and their approximate locations in the human brain. See the main text for explanation.

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 4.3). 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 4.1). It is possible that these same regions implement transfer as well (Section 4.7), 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

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.

## 5.3 Cognitive parsing principles

### *5.3.1 Incrementality*

The linear phase algorithm is an incremental, 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 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 (50), 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.

(50)

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 are stored in a temporary working memory and enter the syntactic component 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 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).

### *5.3.2 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 4.6). 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.

### *5.3.3 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.

### *5.3.4 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 (51) is first attached to the lowest possible solution, and only if that solution fails, to the nonlocal node.

(51) 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 6.5.4). If no choice is provided, bottom-up algorithm is used as default.

### *5.3.5 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 4.3, 6.5.2). 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 6.5.4). 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.

### *5.3.6 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 6.5.4).

### *5.3.7 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 is correct. The user can activate the working memory or knock it off by changing the study parameters in a manner explained in Section 6.5.4 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.

### *5.3.8 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 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 [$\alpha$ $\beta$] 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.

## 5.4 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 6.6.6) 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.

## 5.5 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 4.1) 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.

# 6 Inputs and outputs

## 6.1 Installation and 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 document), *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 or Windows shell. 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 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.

Each trial run that is launched by the above command involves a host of internal parameters that the user can configure. These involve things such as where is the lexicon, test corpus, what heuristic principles should be used, and many others. The information can be provided in several ways. One way is to provide it inside a configuration file called *config_study.txt*. If the parser is launched without any parameters, as in the example

above, then it will try to find this file from the installation directory. Usually this file is present in any version currently being developed. If the file is not found, however, default values will be used. Finally, any parameter that is defined in the *config_study.txt* can also be defined as an input parameter to the program, which will overwrite any specifications found from the configuration file. This makes it possible to control the execution of the script form an external source, say from an external program that one might want to use to organize scripts that perform several studies. Figure 35 shows the contents of the configuration file I have currently on my computer; explanations follow.

```
author: Pauli Brattico
year: 2021
date: April
study_id: 1
study_folder:           language data working directory/study-4_d-LHM/
lexicon_folder:         language data working directory/lexicons
test_corpus_folder:     language data working directory/study-4_d-LHM/
test_corpus_file:       LHM_corpus.txt

only_first_solution: False
logging: True
ignore_ungrammatical_sentences: False
console_output: Full

datatake_resources: True
datatake_resource_sequence: False
datatake_timings: False
datatake_images: False

image_parameter_stop_after_each_image: False
image_parameter_show_words: True
image_parameter_nolabels: False
image_parameter_spellout: False
image_parameter_case: False
image_parameter_show_sentences: True
image_parameter_show_glosses: True

extra_ranking: True
filter: True
lexical_anticipation: True
closure: Bottom-up
working_memory: True

positive_spec_selection: 100
negative_spec_selection: -100
break_head_comp_relations: -100
negative_tail_test: -100
positive_head_comp_selection: 100
negative_head_comp_selection: -100
negative_semantics_match: -100
lf_legibility_condition: -100
negative_adverbial_test: -100
positive_adverbial_test: 100

```

**Figure 35**. Screenshot from the study configuration file *config_study.txt*.

Each line has two fields: the key and a value, separated by ":". The key determines the name of the parameter. For example, the key *test_corpus_folder* determines the name of the parameter that defines the folder from where the program tries to find the test corpus file. It is followed by the name of the folder. Each parameter is read in the same way. Whatever keys and values are provided will be read into a settings data structure (dictionary) that can then be used anywhere in the program. It is not a requirement that the user specifies all keys in this file. If a key or parameter is missing, then that parameter is not used, or a negative value is assumed. If a parameter is missing that is mandatory for normal operation, such as the test corpus file, the program will

attempt to use a default value. If also that strategy fails, then an error message will likely appear. Keys cannot contain white spaces, values can.

The same key-value pairs can be given as input arguments to the function from the command prompt. They are given by writing *key=value*, that is, key followed by "=" followed by the value. For example, if the user wants to run a test corpus from folder /*my_test*, then the following command executes the script with that parameter:

```
python lpparse test_corpus_folder=my_test/
```

Whatever parameters are provided in the command prompt will always override parameter specifications that are given anywhere else (in the study configuration file or by default). This method is useful if the user wants to control the script from an external program or perhaps by another Python script that is used to run several studies ("multistudies").

Recall that the key cannot contain white spaces, but the values can. If the user uses white spaces on the command line, the separated strings will be interpreted as two separate parameters which gives a wrong result. To provide such parameters correctly, the user must use quotation marks as follows:

```
python lpparse "test_corpus_folder=my test folder/"
```

This will treat "test_corpus_folder=my test folder" as one argument.

## 6.2 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.

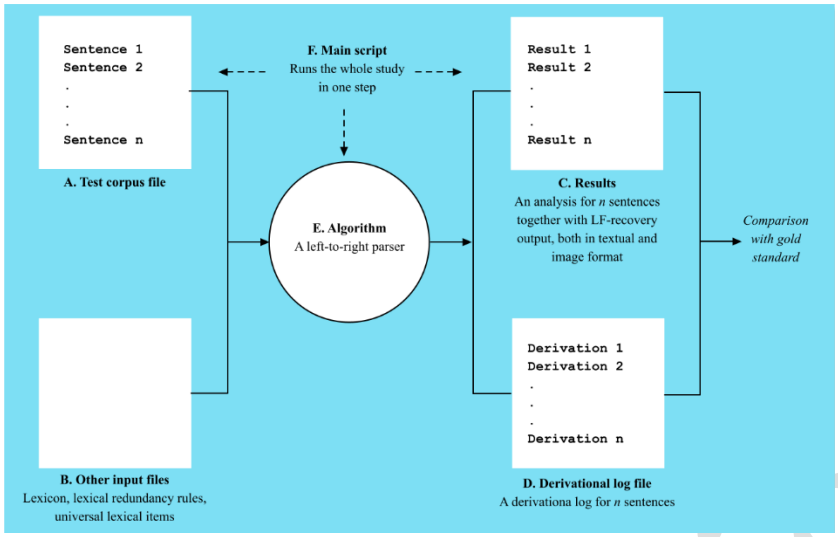



Figure 12. Relationships between the input, main script, linear phase parser and the output.

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 raw data associated with each study. 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 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. |
| lexical_stream.py | Defies properties characterizing the lexical stream which “streams” primitive lexical items and inflectional features from the lexico-morphological component into the syntactic component. |
| 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. |
| narrow_semantics.py | A gateway or "shell" that wraps the syntactic pathway and sends grammatical (grammaticalized) features to various subsystems for interpretation. These interpretations then generate objects into the discourse inventory where they become visible for global cognition. This module maintains the discourse inventory. |
| 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. Does not work currently and will not be needed in the future. |
| 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. |
| SEM_LF-recovery | Performs LF-recovery and is called by narrow semantics. |
| SEM_operators_variables | Interprets operator-variable constructions and populates the discourse inventory accordingly. |
| SEM_pragmatic_pathway | Handles computations involved in the pragmatic pathway that currently computes information structure and grammaticalized discourse features [D:]. |
| 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. |

When the user runs the program by writing *python lpparse* into the command prompt in the root folder of the program, this command will launch the function *__main__.py* inside the *lpparse* folder. This entry point can divert the execution into other modules as defined by input arguments. If we run a standard study, then *main.py* is called by using the arguments provided by the user. 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_study.txt* file is read by *main.py* module. Once the *main.py* knows the location of the test corpus and the study parameters (on the basis of input argument or by reading them from the study configuration file), it will read the sentences.

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." 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.

## 6.3 Running several studies

It is possible to run several studies at once. This can be done by writing a separate script that calls *main.py* with separate arguments.

## 6.4 Main script (*main.py*)

The main script runs one individual study on the basis of parameters provided. It configures the parser on the basis of the study parameters 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.

## 6.5 Structure of the input files

### *6.5.1 Test corpus file (any name)*

The test corpus file name and location are provided in the *config_study.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. Symbols # and ‘ in the beginning of the line is read as a comment and is ignored. The use of ‘ allows the user to write glosses below the test sentences, which is useful if they belong to some other language than English. 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 (input sentences can also be provided from the command prompt). 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).[8] Figure 14 is a screen capture from one test corpus file to illustrate what it looks like.



[8] 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.

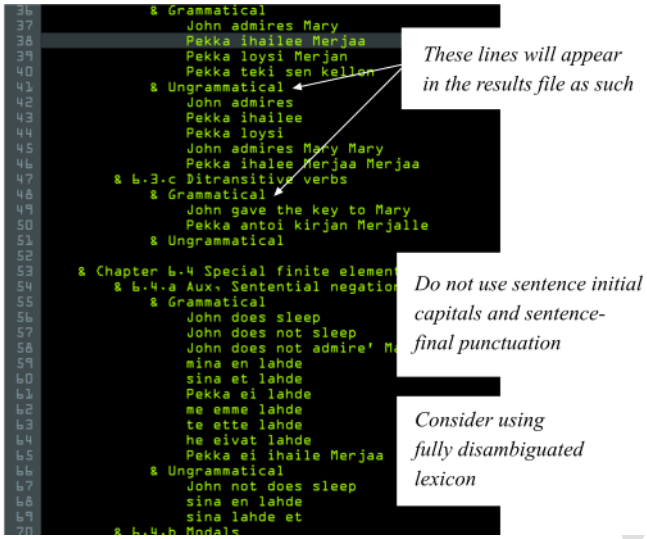



Figure 14. Screen capture of a test corpus file.

### 6.5.2 *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*. The first contains language specific lexical items, the second a list of universal redundancy rules and the last a list of universal morphemes. Figure 15 illustrates the language-specific lexicon.

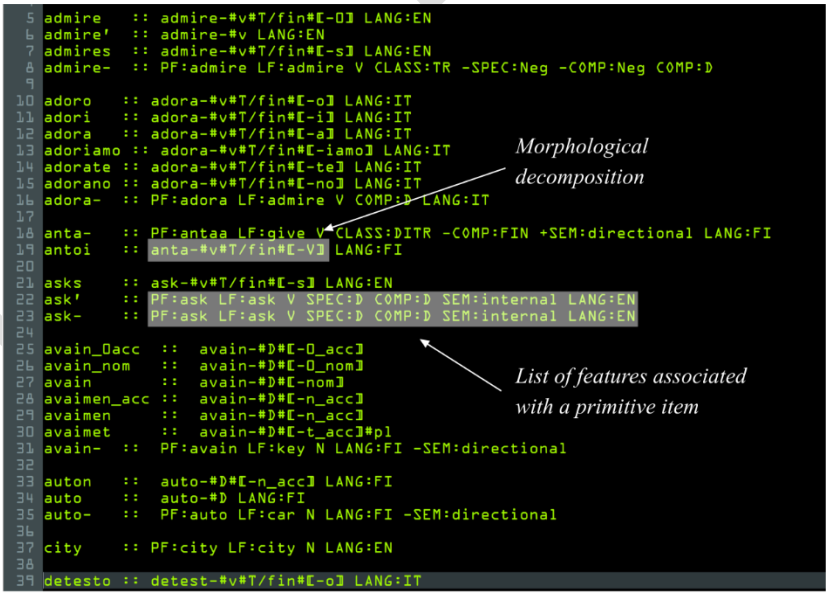



Figure 15. Structure of the lexical file (*lexicon.txt*)

Each line in the lexicon file begins with the surface entry that is matched in the input. This is followed by symbol "::" which separates the surface entry from the definition of the lexical item itself. If the surface entry

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

A lexical feature is a string, a "formal pattern" or ultimately a neuronal activation pattern. They do not have further structure and are processed by first-order Markovian operations. The way any given feature reacts inside syntax and semantics is defined by the computations that process these lexical items and the "patterns" in them.[9] As can be seen from the above screen capture, most features have a 'type:value' structure, where the type dictates the system that processes it, value is the input that will generate a specific interpretation. Figure 16 contains a summary of the most important lexical features that the syntax and semantics reacts in the current model.



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

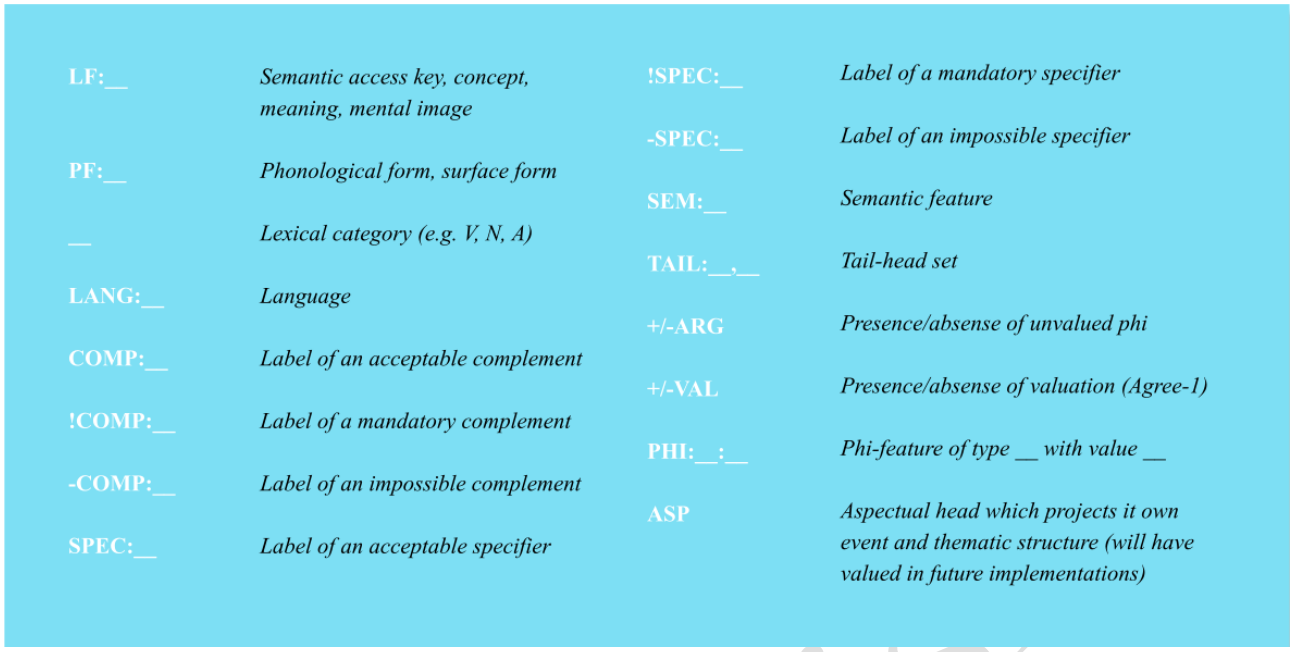



Figure 16. Selection of lexical features

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.

### *6.5.3 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.

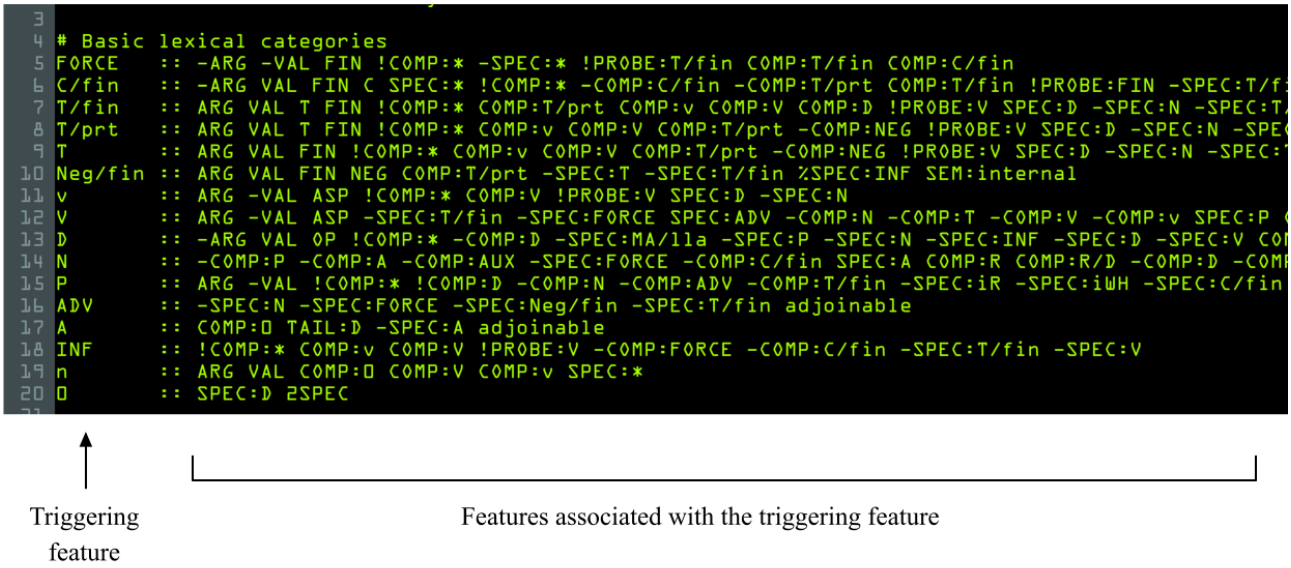



Figure 17. Lexical redundancy rules, as a screen capture from the *redundancy_rules.txt* file.

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. 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).

### *6.5.4 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.

## 6.6 Structure of the output files

### *6.6.1 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.

```
107
108 4. Pekka ihailee Merjaa                                        Input sentence
109
110     [<D Pekka>:1 [T [<__>:1 [v [ihaile- [D Merja]]]]]]          Syntactic analysis
111
112     Semantics:
113     Recovery: ['Agent of v(Pekka)', 'Patient of ihaile-(Merja)']
114     Aspect: []
115     D-features: []                                              Aspects of semantic interpretation
116     Operator bindings: []
117     Speaker attitude: ['Declarative']
118     Information structure: {'Marked topics': ['1'], 'Neutral gradient': ['2'], 'Marked focus': []}
119
120     Resources:
121     Total Time:1250, Garden Paths:0, Memory Reactivation:0, Steps:7, Merge:6, Move Head:7, Move Phrase:2,
122     A-Move Phrase:0, A-bar Move Phrase:0, Move Adjunct:2, Agree:3, Phi:4, Transfer:6, Item streamed into syntax:7,
123     Feature Processing:0, Extraposition:0, Inflection:12, Failed fra[illegible]y:2,          Performance metrics
124     LF test:9, Filter solution:5, Rank solution:3, Lexical retrieval:19, Morphological decomposition:3,
125     Mean time per word:416, Asymmetric Merge:18, Sink:4, External Tail Test:17,
126
127     Discourse inventory:
128     Object 1 ['§Thing']
129         Referring constituent: D
130         Order gradient: 1
131         Reference: [D Pekka]
132         Semantic type: ['§Thing']
133         Operator: False                                         Discourse inventory (semantic objects)
134         Marked gradient: High
135         In information structure: True
136     Object 2 ['§Thing']
137         Referring constituent: D
138         Order gradient: 2
139         Reference: [D Merja]
140         Semantic type: ['§Thing']
141         Operator: False
142         In information structure: True
```

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 results 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.

### *6.6.2 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 the processing and morphological decomposition of the phonological words in the input, application of the ranking principles leading into Merge-1, and 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.

```
#1. Pekka nukkuu
['Pekka', 'nukkuu']

    1. ['Pekka', 'nukkuu']

       Next item: "Pekka". Lexical retrieval...(55ms) Word "Pekka-#D#sg#3p#def#hum#[-nom]" contains mul
       Next item: "hum$". Lexical retrieval...(45ms) Inflectional feature hum$...(50ms) Added ['LANG:FI
       Next item: "def$". Lexical retrieval...(45ms) Inflectional feature def$...(50ms) Added ['LANG:FI
       Next item: "3p$". Lexical retrieval...(35ms) Inflectional feature 3p$...(40ms) Added ['LANG:FI',
       Next item: "sg$". Lexical retrieval...(35ms) Inflectional feature sg$...(40ms) Added ['LANG:FI',
       Next item: "D$". Lexical retrieval...(25ms) Adding inflectional features ['LANG:FI', 'NOM', 'PHI
       Item enters active working memory. Computing semantics for D(IDX:1,QND)...Applying semantic crit

       Next item: "Pekka-". Lexical retrieval...(65ms) Done.(70ms)
       Item enters active working memory.

    2. Consume "Pekka": D + Pekka
       One solution due to sinking.(75ms) Done.
       Results:
           (1) D
       Sinking Pekka into D = D(N)(80ms)
       Next item: "nukkuu". Lexical retrieval...(65ms) Word "nukku-#T/fin#[-V]" contains multiple morph
       Next item: "T/fin$". Lexical retrieval...(65ms) Adding inflectional features ['LANG:FI', 'PHI',
       Item enters active working memory.

    3. Consume "T": D(N) + T
       Working memory operation...1 nodes currently in active memory.
       Filtering and ranking merge sites...Filtering...Done. Ranking...Bottom-up baseline ranking...+Sp
       Results:
           (1) D(N)
       Now exploring solution [D(N) + T]...Transferring left branch D(N)...(80ms) Interpreting [D Pekka
       Result: [[D Pekka] T]...Done.

       Next item: "nukku-". Lexical retrieval...(65ms) Done.(70ms)
       Item enters active working memory.

    4. Consume "nukku": [[D Pekka] T] + nukku
       One solution due to sinking.(75ms) Done.
       Results:
           (1) T
       Sinking nukku into T = [[D Pekka] T(V)](80ms)
```

Input sentence

Morphological decomposition

Lexical item arrives to syntactic module

Merge-1

Figure 20. Screen capture from the log file.

Figure 21 illustrates the transfer operations.

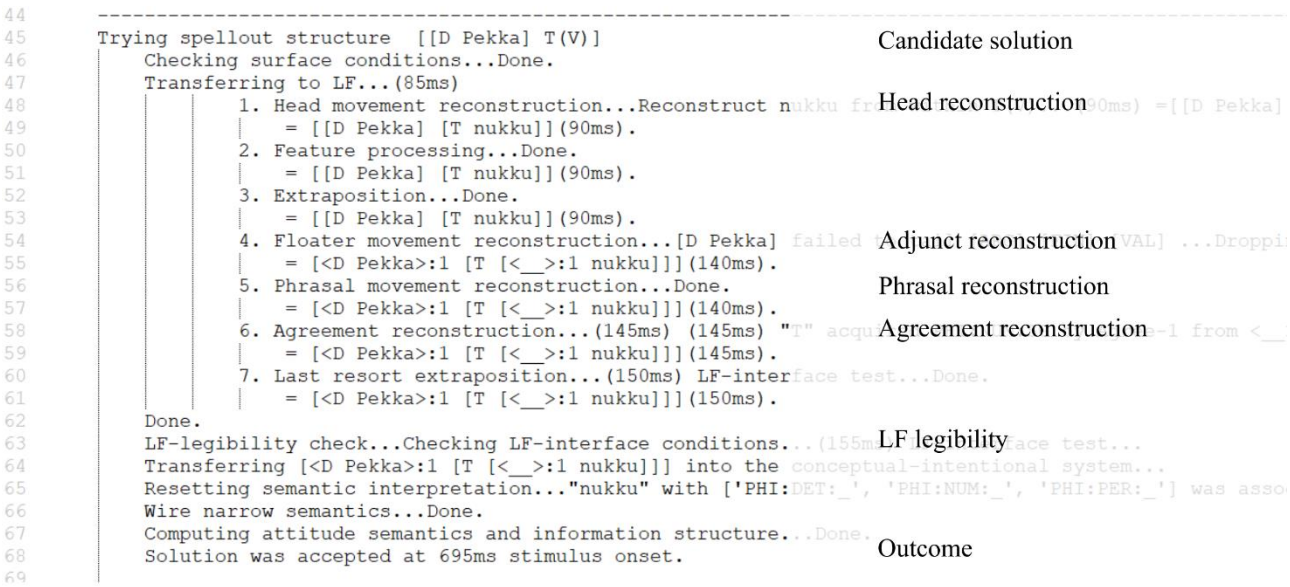



Figure 21. Transfer and evaluation.

### *6.6.3 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.

### *6.6.4 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 6.6.2) and does not appear in this file.

*6.6.5 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 6.5.4). Figure 23 illustrates the phrase structure representation generated for a simple transitive clause in English, when produced without any lexical information.

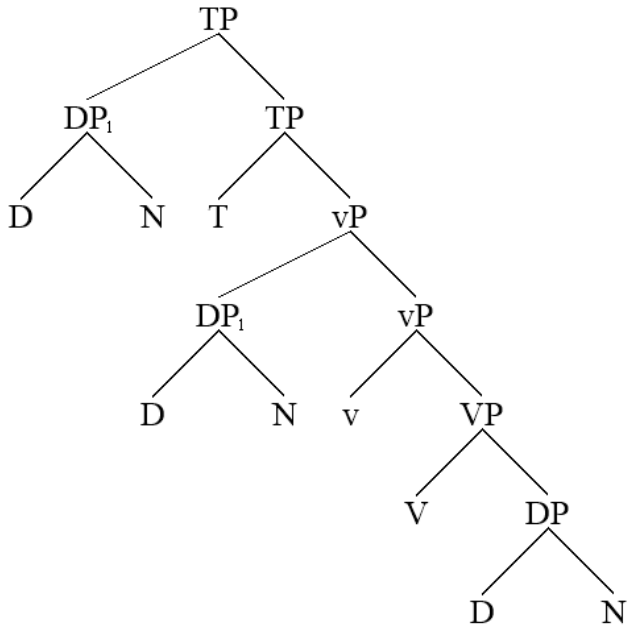



Figure 23. A simple phrase structure image generated by the algorithm for a simple transitive clause *John admires Mary*.

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.

### *6.6.6 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 6.5.1).

# 7 Grammar formalization

## 7.1 Basic grammatical notions (phrase_structure.py)

### *7.1.1 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.

### *7.1.2 Types of phrase structure constituents*

#### *7.1.2.1 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()
```

#### *7.1.2.2 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
```

### *7.1.2.3 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()
```

#### *7.1.2.4 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
```

#### *7.1.2.5 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.

#### *7.1.2.6 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()
```

A (regular) *complement* is defined as the same relation as sisterhood. Geometrical sisterhood plays no relation is complement. We also use 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:*'}
```

#### *7.1.2.7 Labels*

Labeling algorithm is based on (52).

(52) Labeling

Suppose α is a complex phrase. Then

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

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

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

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

```
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_
```

#### *7.1.2.8 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. 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, ⟨β⟩]). *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 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
```

*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
```

#### *7.1.2.9 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]
```

#### *7.1.2.10 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_
```

### *7.1.3 Basic structure building*

#### *7.1.3.1 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
```

#### *7.1.3.2 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).

(53)

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
```

### *7.1.3.3 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
```

#### *7.1.3.4 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
```

### *7.1.4 Nonlocal grammatical dependencies*

#### *7.1.4.1 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:

(54) 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)
```

### *7.1.4.2 Tail-head relations*

A tail-head relation is triggered by a lexical feature (TAIL: $F_1$, … , $F_N$), F… being the feature or set of features that is being searched from a head. In order for F to be visible for the constituent containing the tail-feature, say T, F must occur in a primitive left head H at the upward path from T. An upward path is the path that follows the mother-of relation. 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


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'}
```

## 7.2 Transfer (transfer.py)

### *7.2.1 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 0). The operation targets α and follows a predetermined order: head movement reconstruction → adjunct movement reconstruction → A´/A-movement reconstruction → agreement reconstruction (i.e. Merge-1 → Move-1 → Agree-1).

```
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
```

### *7.2.2 Head reconstruction (head_reconstruction.py)*

Head reconstruction is the first operation performed during transfer. All other components of the transfer presuppose that all grammatical heads have been reconstructed into their correct positions. To model head reconstruction, we must separate two scenarios: one in which (i) the reconstruction targets a complex head in isolation and another in which it (ii) targets a complex phrase that may or may not contain complex heads. The former (i) occurs when the parser considers some solution [α(β) + φ] and we must transfer α(β) in order to evaluate whether [α(β) + φ] is acceptable and/or likely. Transfer will target α(β), but the problem is that this situation involves a branching point between two solutions [[α β] φ] or [α [β φ]]. This requires that we insert amount of 'parsing intelligence' inside head reconstruction that decides the issue. The code is as follows:

```
if ps.is_complex():
    return self.reconstruct_head_movement(ps)

if ps.is_complex_head() and self.left_branch_constituent(ps) and
self.controlling_parser_process.LF_legibility_test(self.reconstruct_head_movement(ps.copy())):
    return self.reconstruct_head_movement(ps)

return ps
```

The first condition triggers standard head reconstruction if the targeted phrase is complex. This will be described below. The second condition is applied if the targeted phrase is itself a complex head α(β). Two further tests are performed: α(β) must be able to appear as a left branch constituent and it must satisfy LF legibility alone as a left branch. If these tests pass, α(β) will be reconstructed in isolation, creating a trivial chain [α(_1) $\beta_1$]. If the tests do not pass, α(β) will not be reconstructed. It will then be encountered again when the hosting phrase is targeted and reconstructed into the right branch, as explained below. A prototypical example of the scenario (i) is [D(N) + T], where we generate [[D N] T]. A prototypical example of (ii) is [T(v,V) + D]. The complex head T(v, V) will be evaluated at this step, but reconstruction will not be attempted since it cannot be reconstructed into a legitimate left branch. The complex head remains in the structure. When the whole clause is later targeted for head reconstruction, it will be correctly expanded into the right branch to generate [T [v [V DP]]]. The right branch expansion is described below and constitutes what we would intuitively consider standard head reconstruction.

Standard reconstruction is performed inside function *reconstruct_head_movement*. The algorithm performs minimal search from the top node, detects complex heads (to the left and bottom right), determines an intervention feature and then creates a head chain:

```
# ------------------ minimal search ----------------------------------------------#
for node in phrase_structure:
    if self.detect_complex_head(node):
         complex_head = self.detect_complex_head(node)
         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()
```

The first condition detecting a complex head is trivial. The intervention feature detection is a heuristic shortcut that replaces something whose fundamental nature is unclear to me. It serves three purposes. First, it allow us to separate local head reconstruction that is HMC-compliant from head reconstruction that is nonlocal. Second, it allows us to capture island or intervention restrictions that regulate the latter. And third, due to its relativized nature it makes room for other options ('semi-nonlocal head movement') that to me seem possible in the light

of crosslinguistic facts. The point, however, is to restrict head reconstruction into certain grammatical domains depending on the feature content of the complex head. It is inspired by Luigi Rizzi's relativized minimality approach. The third function creates the actual head chain, which we examine next.

Let us consider the creation of the head chain next. The function first considers the possibility that we have a complex head but no further structure into which it could be reconstructed. This happens, for example, if the bottom right constituent is D(N). Then we reconstruct N into the sister of D to create [α D(N)] → [α ... [D N]].

```
if self.no_structure_for_reconstruction(complex_head):
    self.reconstruct_to_sister(complex_head, affix)
```

The only nontrivial property of this operation is that we must handle a situation in which the reconstructed head is still not primitive. In such case head reconstruction is called recursively from the *reconstruct_to_sister* function. The minimal search operation that creates the head reconstruction chain is as follows.

```
# ------------------------------------------------------------------------------------------------------#
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):
        self.controlling_parser_process.consume_resources("Move Head")
        return
    affix.remove()
# ------------------------------------------------------------------------------------------------------#
```

The first condition takes care of the intervention, relying on the intervention feature defined earlier and terminates the process if intervention is detected. The ultimate reason for intervention remains unknown. Termination leads into the use of the local last resort strategy (function *last_resort*). If there is no intervention, the operation tries to merge the head into the structure into position [X αP] and then tests if the position will be accepted. If it is accepted, the head will remain in that position and the operation terminates. If the position is not accepted, the affix is removed, and minimal search continues. For the position [X αP] to be accepted, two conditions must be satisfied:

```
if not self.head_is_selected(affix):
    return False
if not self.extra_condition(affix):
    return False
```

The first is that the head must be selected "from above," and the extra condition refers to a special situation that only occurs in connection with the left peripheral C-T system. The latter concerns the problem of how to force head reconstruction to respect the possible EPP feature of T and reconstruct below the phrase that will end up checking the EPP feature. We pick the first position that satisfies these conditions, and then the search terminates.

If minimal search reaches the end and there is still no solution, then in most situations the system will use the last resort option. There is one more edge case to take care of, namely a situation in which the last remaining node is complex while no solution has been found for a head that comes from above. The reason this situation must be attended because it could be that the node that is required for selection of the reconstructing head is still hiding inside the complex right bottom node. The operation begins by reconstructing the bottom complex head if it is complex. Then the operation branches on the basis of the head of the element. Suppose the reconstructing head is α. If the bottom node was D(N) and now [D N], we try [DP α]. In it is not a DP, we apply [X α], X = bottom node after reconstruction, with the exception that if X has the intervention feature, the solution is [α X], thus we respect intervention. Then the result is evaluated and, if it is accepted, nothing is done; if it still rejected, then the last resort solution will be used.

```
if 'D' in node.mother.head().features:
    node.mother.merge_1(affix, 'right')
else:
    node = node.top().bottom()
    if intervention_feature not in node.features and intervention_feature not in node.sister().features:
         node.merge_1(affix, 'right')
    else:
         node.merge_1(affix, 'left')
if self.reconstruction_is_successful(affix):
    self.controlling_parser_process.consume_resources("Move Head")
    return True
```

### *7.2.3 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:

(55) *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:

(56) *Fitting a floater*

α can be fitted into position P if and only if

(i) tail-head features are satisfied (Section 7.2.4),

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

(iii) P is not a local specifier position.

```
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).

```
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.

```
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.

```
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()
    # ----------------------------------------------------------------------------------------#
```

### *7.2.4 External tail-head test*

External tail-head is defined in the following way.

(57) 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.

### *7.2.5 A´/A-reconstruction Move-1 (phrasal_reconstruction.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:

(58) *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 [$\alpha_{EPP}$ XP]] and βP has not been moved already. If βP has no criterial features, it P will undergo A-reconstruction (local successive-cyclicity, Section 7.2.6); 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 7.2.7).

### *7.2.6 A-reconstruction*

A-reconstruction is an operation in which a DP makes a local spec-to-spec movement, i.e. [$DP_1$ [α $\_\__1$ β]]. The operation is implemented if and only if 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
```

### *7.2.7 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\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\t\tThe structure is still uninterpretable.')
        # ------------------------------------------------------------------------------------------#
        log(f'\t\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].[10]

```
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():
```

[10] 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.

```
                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 7.2.8). 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:

| | | | | | | |
|---|---|---|---|---|---|---|
| (59) | *John* | *gave* | | *the book* | *to* | *Mary* |
| | [John | [T | [$_{DP}$[ | the book | [P | Mary]]] |
| | | | | SPEC | P | COMP |

This interpretation is wrong. First, it contains a preposition phrase [*the book* P DP], which is ungrammatical in English (by most analyses). Second, the verb *gave* now has the wrong complement PP when it required a DP. Extraposition will be tried as a last resort, in which it is assumed that the string 'D + N + [P + D + N]' should be interpreted as [DP ⟨HP⟩]. This will fix both problems: the verb now takes the DP as its complement (recall that the right adjunct is invisible for selection and sisterhood), and the preposition phrase does not have a DP-specifier. It is easy to see how the PP satisfies the criteria for the application of the extraposition operation: PPs are adjoinable, the configuration is [DP PP], and the PP is inside a FinP. The final configuration that passes LF-legibility test is (60).

(60) [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*'.

### *7.2.8 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.

```
# 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)
```

### 7.3 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():
        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 2019b). 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
```

## 7.4 LF-legibility (LF.py)

### *7.4.1 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)
```

### *7.4.2 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).

### *7.4.3 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.

## 7.5 Semantics (narrow_semantics.py)

### *7.5.1 Introduction*

Semantic interpretation is implemented in the module narrow_semantics.py. It contains functions which read information arriving from the syntactic pathway as 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 core of the operation is defined by a recursive interpretation function, which provides a "router" that calls individual submodules on the basis of the lexical features.

```
def _interpret(self, ps):
    if ps.is_primitive():
        self.LF_recovery_module.perform_LF_recovery(ps, self.semantic_interpretation)
        self.detect_phi_conflicts(ps)
        self.interpret_tail_features(ps)
        self.operator_variable_module.bind_operator(ps, self.semantic_interpretation)
        self.pragmatic_pathway.reconstruct_discourse(ps, self.semantic_interpretation)
        if self.failure():
            return
    else:
        if not ps.left_const.find_me_elsewhere:
            self._interpret(ps.left_const)
        if not ps.right_const.find_me_elsewhere:
            self._interpret(ps.right_const)
```

An important feature of this mechanism is its recursive character. It will examine the whole phrase structure object (ignoring copies left after reconstruction) and react to any feature requiring interpretation.

### *7.5.2 Operator-variable interpretation (SEM_operators_variables.py)*

The operator-variable module interprets operator-variable constructions and populates the discourse inventory accordingly. Each operator [OP:F] is paired with a matching scope marker [OP:F][FIN], the latter which "denotes" the corresponding proposition. The kernel of the module is constituted by a function that binds operators with their propositional scope markers. First, operator features that are part of C are ignored; if they are included, then we get a list of self-referring operator constructions that might be posited for some reason but where removed in order to prevent redundant and misleading information from appearing in the output files.

```
def bind_operator(self, operator_ps, semantic_interpretation):
    if 'C' in operator_ps.features or 'C/fin' in operator_ps.features:
        return
```

Next, all operator features of the lexical item are targeted for processing and are bind by a scope marker.

```
feature_set = operator_ps.head().features.copy()
for f in feature_set:
    if self.is_operator_feature(f):
        scope_marker_lst = self.bind_to_propositional_scope_marker(operator_ps, f)
```

If the binder is not found, then the interpretation fails. If it is found, the proposition marked by the propositional scope marker is generated into the discourse inventory if it does not already exist. Binding information is also added to the discourse inventory. This allows other cognitive processes to access it.

```
        if not scope_marker_lst:
            self.interpretation_failed = True
            break
        else:
            semantic_interpretation['Operator bindings'].append((f'{operator_ps.illustrate()}',
             f'{scope_marker_lst[0]}[{f}]'))
            idx = self.narrow_semantics.get_semantic_wiring(operator_ps)
            if not idx:
                self.narrow_semantics.wire(operator_ps)
                idx = self.narrow_semantics.get_semantic_wiring(operator_ps)
            self.narrow_semantics.update_semantics_for_attribute(idx, 'Bound by', scope_marker_lst)
            self.interpret_and_update_operator_feature(idx, f)
            if not self.narrow_semantics.controlling_parsing_process.first_solution_found and not
                 self.narrow_semantics.get_semantic_wiring(scope_marker_lst[0]):
                if self.full_proposition(scope_marker_lst[0]):
                    self.narrow_semantics.wire(scope_marker_lst[0])
```

### *7.5.3 Pragmatic pathway*

This section remains to be filled (the study is currently under revision).

### *7.5.4 LF-recovery (SEM_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 $\alpha$ must satisfy two conditions: $\alpha$ cannot be a copy of an element that has been moved elsewhere and $\alpha$ 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.

```
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\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
```

# 8 Formalization of the parser

## 8.1 Linear phase parser (linear_phase_parser.py)

### *8.1.1 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.

### *8.1.2 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. The function will first check if there is any reason to terminate processing (and hence, escape the recursion before any further steps are taken). Processing is terminated if there are no more words, or if a self-termination flag *self.exit* has been raised somewhere during the execution.

```
if self.circuit_breaker(ps, lst, index):
    return
```

If there are no more words, α will be send for interpretation. This function is *complete_processing*. It implements several operations (transfer, LF-legibility, semantic interpretation) discussed later in this subsection. Suppose word w was consumed. To retrieve properties of w, lexicon will be accessed.

```
list_of_retrieved_lexical_items_matching_the_phonological_word = self.lexicon.lexical_retrieval(lst[index])
```

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.

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.

```
terminal_lexical_item, lst_branched, inflection = self.morphology.morphological_parse(self,
lexical_constituent, lst.copy(), index)
```

The new list therefore contains the morphological decomposition of the word. These two functions implement the *lexical-morphological module* of the theory. The first item in this list, as provided by all the above step, will then be send to the syntactic component via *lexical stream*.

```
terminal_lexical_item = self.lexical_stream.stream_into_syntax(terminal_lexical_item, lst_branched,
inflection, ps, index)
```

The lexical stream pipeline handles several operations. For example, some of the morphemes in word w could be inflectional, in which case they are stored as features into a separate memory buffer inside the lexical stream 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. The lexical input stream does not contain phonological words, but lexical items and features. A complex phonological word such as *pudo-t-i-n-ko-han* will enter syntax in the form (61). Notice that the order of the morphemes is reversed.

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

fall-cau-past-1sg-Q-hAn → (decomposition)

hAn * Q * 1sg * T * v * V (reverse order into syntax, lexical input stream)

The items then enter the *syntactic module*. Suppose morpheme *hAn* was consumed. The morpheme will be merged to the existing phrase structure in the syntactic working memory, 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.

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

Each site from the ranking is explored. If a ranked list has been exhausted without a legitimate solution, the algorithm will backtrack to an earlier step.

```
# ------------------------------------------------------------------------------------------#
# Examine each possible solution
for site in merge_sites:
    # Target left branch (involves copying for recursive backtracking purposes)
    left_branch = self.target_left_branch(ps, site)
    # Attach the new item to the partial phrase structure at [left branch]
    new_constituent = self.attach(left_branch, site, terminal_lexical_item)
    # Call for next item
    self.put_out_of_working_memory(merge_sites)
    self.parse_new_item(new_constituent, lst_branched, index + 1)
    if self.stop_looking_for_further_solutions():
         break
# ------------------------------------------------------------------------------------------ #
```

All these steps are illustrated in Figure 36.

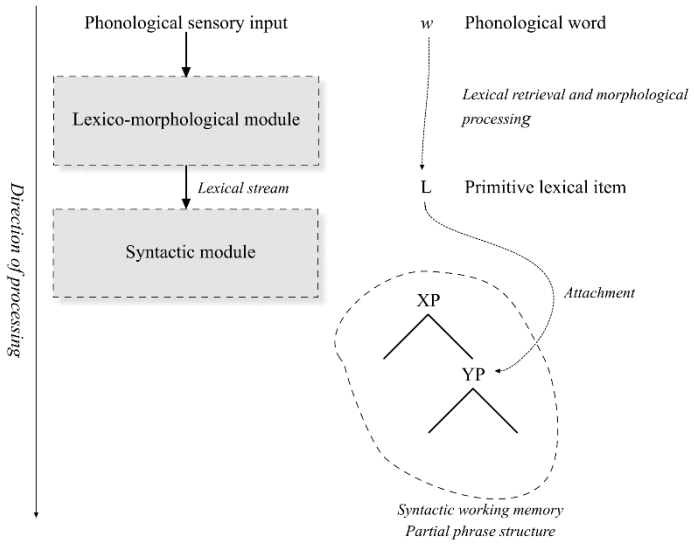



**Figure 36**. A schematic illustration of the early phases of the parser loop.

Once all words have been processed (no more words remain in the input), the result will be submitted to a finalization stage implemented by function *complete_processing*. This process will apply transfer, LF-legibility and semantic interpretation to the top node, but it does several other things as well. First it tests that

all surface conditions are satisfied. Surface conditions are conditions that a well-formed spellout structure must satisfy.

```
if self.surface_condition_violation(ps)
    self.add_garden_path()
    return
```

It is not obvious that surface conditions are needed. Currently these involve only conditions regulating clitics, which must observe certain properties regulating the syntax-phonology mapping. The issue is empirically nontrivial. If the surface conditions are satisfied, the whole phrase structure object will be transferred and then checked for LF-legibility and semantic interpretation:

```
ps_ = self.transfer_to_LF(ps)
if self.LF_condition_violation(ps_) or self.narrow_semantics.interpret(ps_):
    [...] return
```

If it passes these two filters and is the first solution found, an attempt will be made to interpret the sentence against its discourse context by utilizing operations inside the pragmatic pathway.

```
self.narrow_semantics.pragmatic_pathway.compute_speaker_attitude(ps)
self.narrow_semantics.pragmatic_pathway.compute_information_structure(ps)
```

Thus, only completed input sentences are evaluated against the overall communicative situation. The rest of the operations performed inside this function are (imporant) logging operations.

## 8.2 Psycholinguistic plausibility

### *8.2.1 General architecture*

The parser component obtains a list of all possible attachment sites given some existing partial phrase structure $\alpha$ and an incoming lexical item $\beta$. 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.

### *8.2.2 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.

### *8.2.3 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
```

### *8.2.4 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.

### *8.2.5 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.

## 8.3 Morphological and lexical processing (morphology.py)

### *8.3.1 Introduction*

The model 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.

### *8.3.2 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_)
```

## 8.4 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 $\alpha$ is evaluated during [$\alpha$ $\beta$], the operations are counted irrespective of whether $\alpha$ 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.

# 9 Working with empirical materials

## 9.1 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.

## 9.2 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.

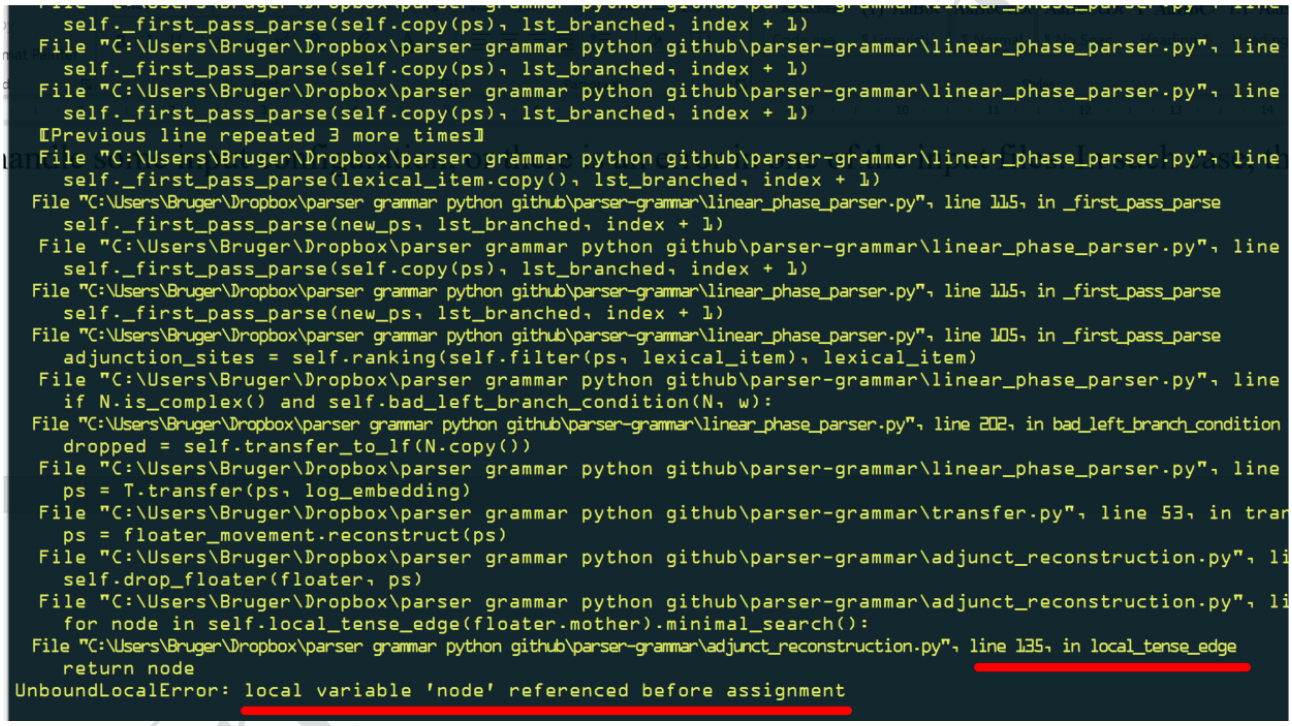



Figure 25. An error report printed to the console. The two most important pieces of information are underlined: the location where the error was encountered (module *adjunct_reconstruction.py, line 135*) and the type of the error.

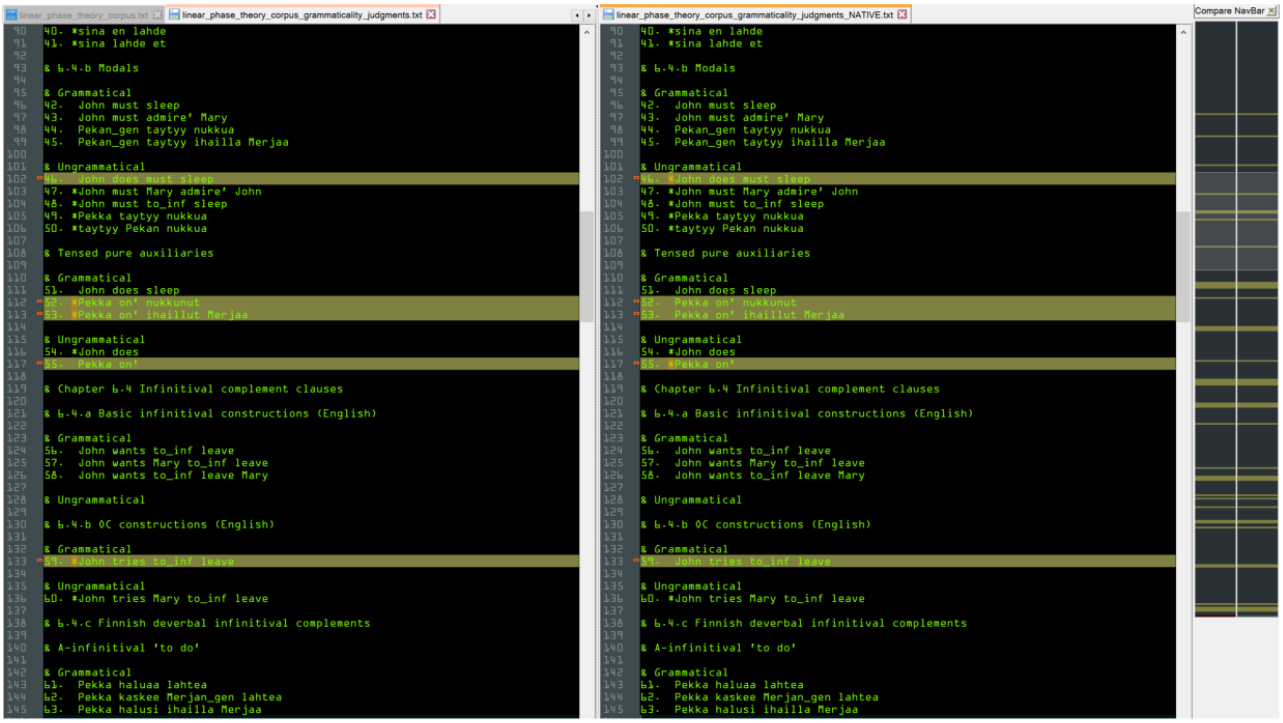
You would then navigate to the indicated location (*adjunct_reconstruction.py, line 135*) and examine the source of the problem. In some rare cases the error is trivial to fix, but in most cases, it is not. This is because the position at which the error is encountered in run-time is not typically the 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.

## 9.3 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 (62) in Finnish. The algorithm judged the sentence ungrammatical.

(62) Ketä Pekka ihailee?
who.par Pekka.nom admire.3sg
'Who does Pekka admire?'

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.

## 9.4 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.

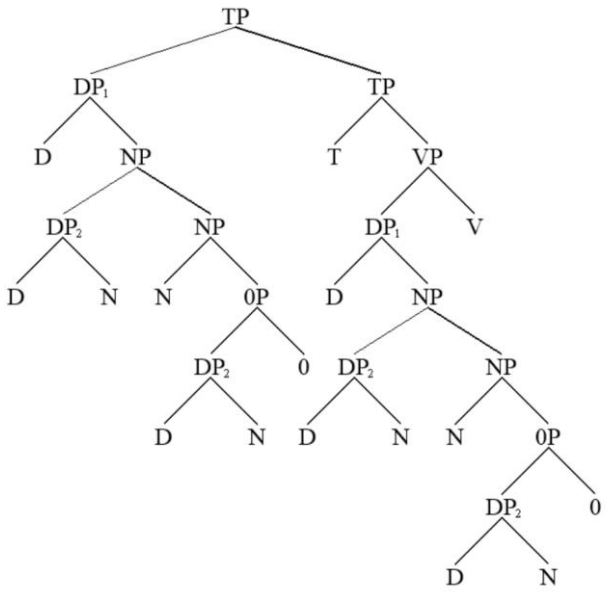



Figure 27. A bare phase structure image.

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*.

### 9.5 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. "Finnish Word Order and Morphosyntax." *Manuscript*.

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

Brattico, Pauli. 2020. "Finnish Word Order: Does Comprehension Matter?" *Nordic Journal of Linguistics* 44(1):38–70.

Brattico, Pauli. 2021. "Case Marking and Language Comprehension: A Perspective from 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*.

Chomsky, Noam. 1964. *Current Issues in Linguistic Theory*. Mouton.

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.

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.