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

(version 15.0)

2019

(Revised June 2023)

Pauli Brattico

Abstract

This document describes a computational implementation of a linear phase parser. 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 15.0). 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 interpreted as a realistic description of the information processing steps involved in real-time language comprehension. It captures both cognitive principle of language ("competence") and cognitive mechanisms involved in language comprehension and use ("performance").

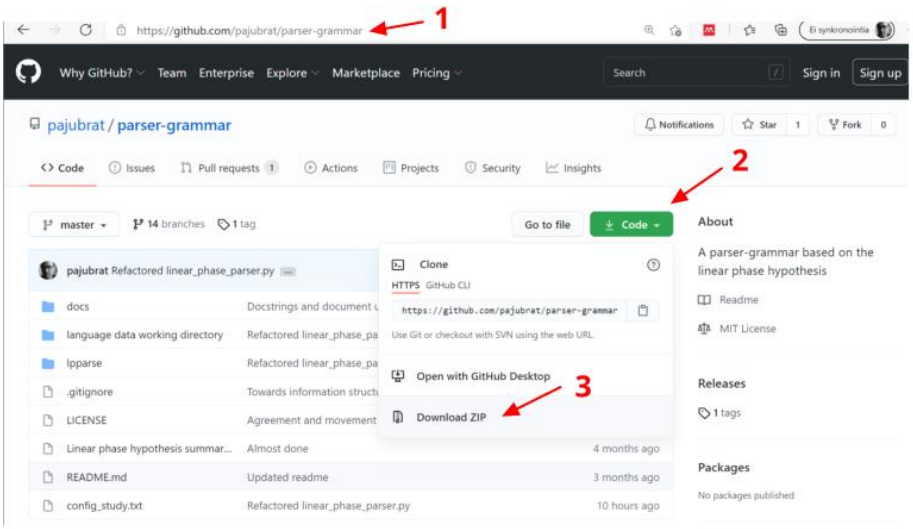
This document describes properties of the version 15.0, which keeps within the framework of the original work but provides improvements, corrections and additions. There may be mismatches between what is described here and what appears in the latest version of the source code. This is because the documentation lags behind and/or has not been completed due to the experimental nature of changes and additions that do not warrant documentation. In addition, some parts of this document are in better condition than others, and there are gaps awaiting documentation. 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 use

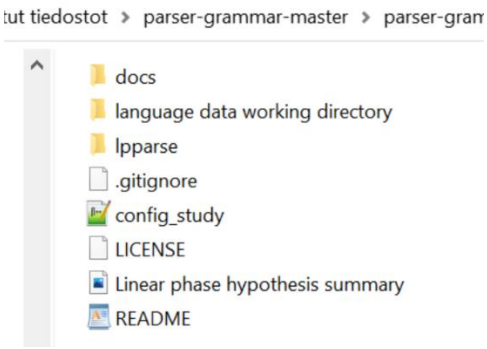
2.1 Installation

The linear phase comprehension algorithm is a collection of Python functions that processes natural language sentences by using principles of human linguistic competence and performance. It works by reading test sentences from a dataset (test corpus) file and analyzing them. The program can be installed on a local computer by cloning it from the source code repository *https://github.com/pajubrat/parser-grammar*. The easiest method is by downloading the software package as one ZIP file and extracting it into a directory in the local machine. Navigate to the source repository by using any web browser (1, Figure below), then click the button “Code” (2) and select “Download ZIP” (3).



Once the package is on the local machine, it must be unpacked into any directory. The same files and folders displayed on the above figure should appear in that directory. The script is ready to use. Because the script is written in Python (3x) programming language, the researcher must have

Python installed on the local machine.[2] After installation, the local installation directory should contain the following files and folders:



Folder */docs* contains documentation (e.g., this document and previous versions), */language data working directory* stores the input and output files associated with any particular study, and */lpparse* stores the modules containing the program code. The file *config_study.txt* is a text file that is used to parametrize the operation of the script, in a manner explained below.

2.2 Use

The script is launched by having the Python interpreter to run the program script. In most cases the version copied or cloned from the code repository comes in a working configuration, it should do something meaningful out of the box.[3] To execute the script in Windows, start a command prompt program such as Windows PowerShell or the command prompt ("cmd") and navigate to the local installation directory. The script can be executed by command *python lpparse*. The program reads a study configuration file *config_study.txt* from the local installation directory

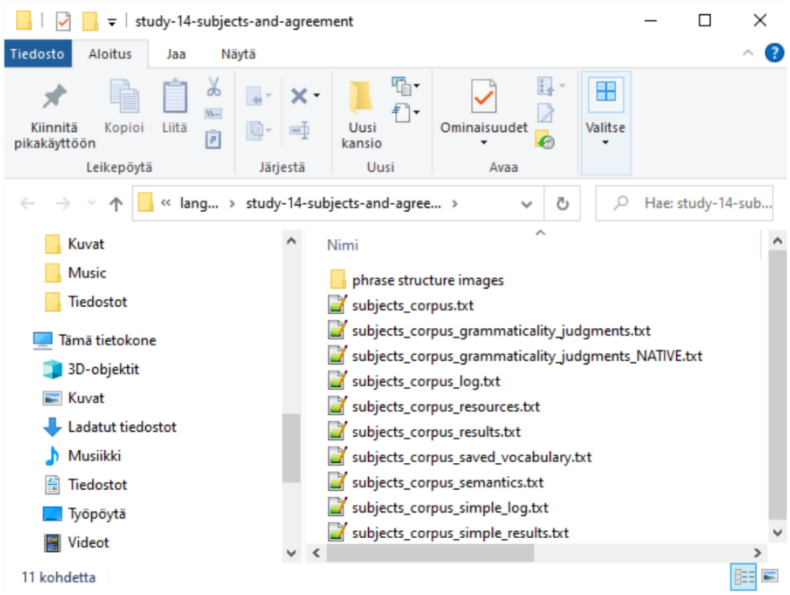
[2] Follow the instructions from *www.python.org*. In Windows, Python installation path must be provided in the Windows PATH environmental variable. Search for the term "setting windows PATH variables." Some versions rely on external Python libraries which provide auxiliary functionality, such as phrase structure tree images and numerical analyses (pandas, matplotlib, numpy).

[3] The version downloaded by following the instructions above 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.

containing the parameters used in the simulation trial. One of these parameters is the folder and name of the *test corpus file* that contains the sentences the script will analyze. See below.

```
    & lähellä 'near' (Class I, IIa)

        Seine saavuttaa meren virtaamalla kenen lähellä
        'Seine.nom reach.prs.3sg ocean.acc flow.ma/inf [who.gen near __]'

        Seine saavuttaa meren kenen lähellä virtaamalla
        'Seine.nom reach.prs.3sg ocean.acc [who.gen near __] flow.ma/inf __]'

        virtaamalla kenen lähellä* Seine saavuttaa meren
        '[flow.ma/inf who.gen near] Seine.nom reach.prs.3sg ocean.acc __'

        kenen lähellä virtaamalla Seine saavuttaa meren
        '[[who.gen near __] flow.ma/inf] Seine.nom reach.prs.3sg ocean.acc __'

    & ali/yli 'under/over' (Class IIb)

        Seine saavuttaa meren virtaamalla ali minkä
        'Seine.nom reach.prs.3sg ocean.acc flow.ma/inf [under what.gen]'

        Seine saavuttaa meren virtaamalla minkä ali
        'Seine.nom reach.prs.3sg ocean.acc flow.ma/inf [what.gen under __]'

        ali minkä virtaamalla Seine saavuttaa meren
        '[[under what.gen] flow.ma/inf __] Seine.nom reach.prs.3sg ocean.acc __'

        minkä ali virtaamalla Seine saavuttaa meren
        '[[what.gen under __] flow.ma/inf __] Seine.nom reach.prs.3sg ocean.acc __'

& Ungrammatical

    Seine saavuttaa meren virtaamalla lähellä kenen
    'Seine.nom reach.prs.3sg ocean.acc flow.ma/inf near who.gen'

    Seine saavuttaa meren lähellä kenen virtaamalla
    'Seine.nom reach.prs.3sg ocean.acc near who.gen flow.ma/inf'

    virtaamalla lähellä kenen Seine saavuttaa meren
    'flow.ma/inf near who.gen Seine.nom reach.prs.3sg ocean.acc'

    lähellä kenen virtaamalla Seine saavuttaa meren
    'near who.gen flow.ma/inf Seine.nom reach.prs.3sg ocean.acc'
```

This is a screenshot of the dataset (a list of sentences with comments) the script processes. The results are generated into several files inside the study folder. This is what I see after running the script on the current configuration on my local machine:



The first file is the test corpus, followed by several outputs generated by the algorithm. For example, the file *subjects_corpus_grammaticality_judgments.txt* contains a list of the original sentences and the grammaticality judgments provided by the model. These files can be opened by any text editor. The folder */phrase structure images* contains a phrase structure image for each grammatical input sentence, as calculated by the model[4]:

[4] In order to generate these images, the user must install the pyglet library and set image generation on in the configuration file.

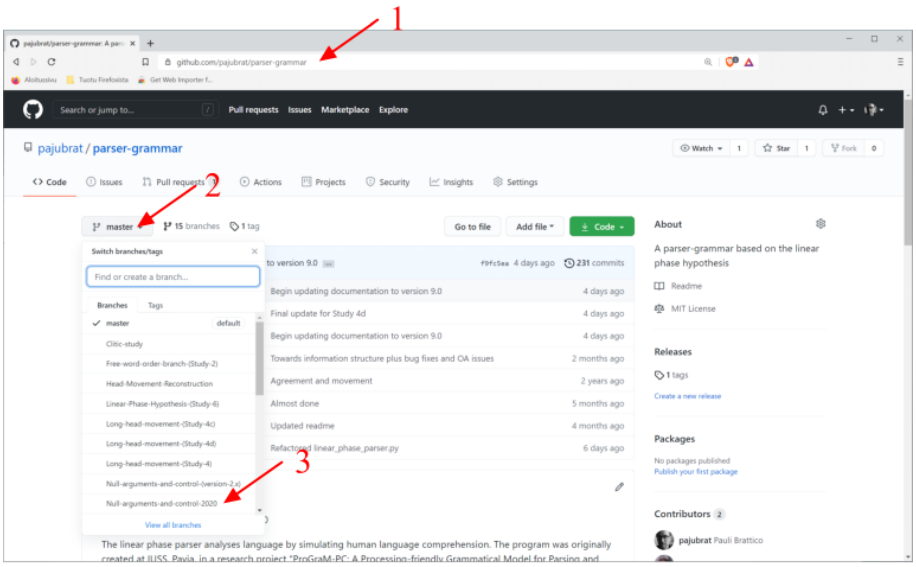
2.3 Structure of the script

When the user runs the script by typing *python lpparse* inside the installation director, the Python interpreter will run a main script (called *__main__.py*) from the folder */lpparse*. This folder contains the modules defining the theory and the behavior of the whole system. The *__main__.py* module will call the program code that exists inside the individual files. These files (also called modules) correspond closely to the contents of the empirical theory that they implement. For example, the module *SEM_narrow_semantics.py* contains operations that correspond to an empirical component carrying the same name in the theory. These files can be opened and edited by any text editor. Thus, if the researcher wants to find out how some grammatical operation, principle or module operates, this information can be found from these files. 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.[5]

2.4 Replication

When a study is published, the input files plus its source code is stored in source code repository. 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 previous versions. 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 scripts 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).

[5] This is not literally true. In most instances the interpreter can take advantage of precompiled files. The important point is that any change into the text files specifying the program behavior will automatically change the behavior of the system.



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. We assume that efficient and successful language comprehension is possible from contextless and unannotated sensory input.[6]

Some sensory inputs map into meanings that are hard or impossible to construct (e.g., *democracy sleeps furiously*), while others are judged as ungrammatical. A useful theory of language comprehension must appreciate these properties. The model 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 is said to be *observationally adequate*. Any scientific theory must be, minimally, observationally adequate, and the fact that it is observationally adequate must be shown by deductive calculation. Because the model judges input sentences, it implements a *recognition grammar* (see §3.4).

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

[6] Some aspects of semantic interpretation, such as the intended denotations of pronouns, depend on the context. If no context is present, the sentence receives an all-new reading.

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 sukka-a* 'three.sg.0 sock.sg.par'). The latter are acquired from the input during language acquisition. Universal properties plus the storage systems constitute the fixed portion of the parser, whereas the 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*.

It is possible to design an observationally adequate comprehension theory for Finnish such that it replicates the responses elicited from native speakers, yet the same model and its principles would not work in connection with a language such as English, not even when provided with a fragment of English lexicon. We could then 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 sense.

Because speakers of different languages differ from each other, we need to use different *speaker models* to model the language processing and recognition grammars for different languages, and more generally for different speakers. A speaker model is a combination of the fixed, universal properties of human language capacities and individual properties acquired during language

comprehension. An explanatory adequate grammar will posit just enough fixed universal structure and individual variation to capture the properties of all languages. This could then be accompanied by a formal learning theory which creates speaker models from linguistic and other experiences.

Suppose we have constructed a theory of language comprehension that 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 human brain to automatize recognition and processing of linguistic stimuli in a way that an observationally adequate explanatory model might or might not be concerned with. Furthermore, real language processing is sensitive to top-down and contextual effects that can be ignored when constructing a theory of 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 model engages in astronomical garden pathing when no native speaker exhibits such inefficiencies, the model can be said to be insufficient in its ability to mimic real language comprehension. We say that if the model's computational efficiency and performance behavior is in line with evidence from real speakers, it is *psycholinguistically adequate*. I will adopt this criterion as well. Psycholinguistic adequacy cannot be addressed realistically without addressing observational and explanatory adequacy first, but the vice versa is not true.

Language comprehension is viewed in this study as a species of cognitive information processing that processes information beginning from linguistic sensory input towards semantic interpretations. The framework closely resembles that of (Marr, 1982). Information processing will be modeled by utilizing *processing pathways*. The 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 are presented in an unambiguous linear order.

The output contains a set of semantic interpretations. 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. One way around this problem, adopted here, is to focus on selected aspects of semantic interpretation and try to predict them on the basis of the input. For example, we could decide to focus on the interpretation of thematic roles and require that the language comprehension model provides for each input sentence a list of outputs which determine which constituents appearing in the input sentence has which thematic roles. For example, we could require that the model provides that *John admires Mary* is processed so that John is judged the agent, Mary the patient, 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 on its own way and 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 day 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 (ultimately a collection of neuronal connections) at which all phonological, morphological and syntactic information processing has been completed and semantic processing takes over. 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.[7]

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 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 a few operations. Meaning would then be read off from relatively shallow 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,

[7] Almost any model can be interpreted in these terms. If we assumed 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 morphological decompositions, morphosyntactic structure or part-of-speech (POS) annotations.

then, we do not posit further restrictions or properties to the syntax-semantic interface apart from its existence.

3.2 Computational linguistics methodology

Scientific hypotheses must be justified by deducing the observed facts from the proposed hypothesis. The method is routinely used all advanced sciences. To do this, 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 then addressed 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 present 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.[8]

The aim, then, is to provide a mathematical formula that is both sufficient and necessary to deduce 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).[9] Although it is hard or impossible to show by relying on completely objective criteria that the hypothesis A is the simplest formula possible (in some sense), we can compare two

[8] We are therefore not interested in natural language parsing in the technical engineering sense or in the discovery of correlations by data mining. The purpose of the model is to justify linguistic hypotheses against linguistically and behaviorally relevant datasets.

[9] These concerns have played a major role in recent minimalist grammars. 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."

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 table-lookup model where each input is paired by brute force with the correct output while hypothesis B relies on a general mechanism or rule, B will be voted as the favorite.

Use of computational, automatized methodologies brings an additional benefit that plays a major role in the present work. When the theory and data are aligned computationally, it becomes possible to work with large, standardized datasets instead of each researcher picking up examples and datapoints more or less randomly, or at the very least without any explicit, stated method for selection. For example, when working with a linguistic phenomenon such as case assignment it becomes possible to compose a standardized dataset that captures the essence of phenomenon in a gapless way, as agreed by the community of specialists, and which then forms the baseline against which all theories are compared. A group of language-specific case assignment datasets can be further put into a composite sets that can serve as a baseline for theories that aim for explanatory adequacy. In this way, linguists can eliminate biases in data selection and compare theories more objectively. This method, too, is used in almost all advanced sciences.

3.3 Overview of the hypothesis

This subsection provides a nontechnical introduction to the empirical hypothesis underlying the algorithm. 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.

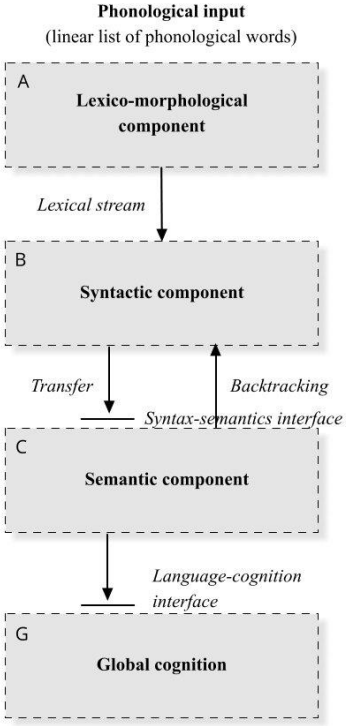


Figure 1. Main components of the model.

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. It is also possible to feed the model directly with lexical items and bypass the phonology-LI mapping. The syntactic component (B) attaches the incoming lexical items incrementally into a partial phrase structure representation in the current syntactic working memory that has been assembled on the basis of the words seen so far. This process is illustrated in Figure 2.

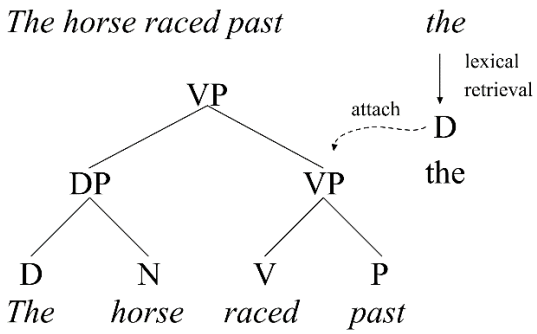


Figure 2. Operation of the syntactic component

The resulting phrase structure representation is called *spellout structure* since it corresponds transparently to the linear order in the sensory input, being directly generated form it.[10] 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 (called narrow semantics) which provides it with detailed semantic interpretation and interacts with global cognitive processes (thinking, decision making, discourse). This corresponds to a process in which the hearer understands the input sentence within some communicative context. If the candidate solution is not grammatical and/or cannot be interpreted semantically, it is *rejected* and no semantic interpretation results. 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. If no acceptable solution emerges, the sentence is judged ungrammatical.

[10] 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.

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 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]

Once an acceptable representation arrives at the syntax-semantics interface, it is interpreted semantically. 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.[11]

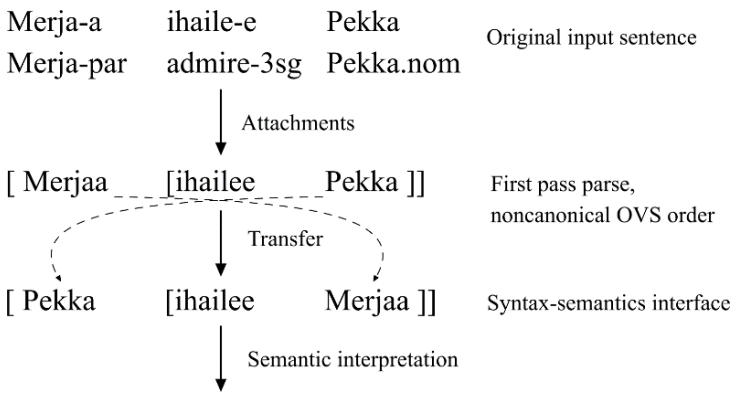


Figure 3. Transfer as an error correction mechanism.

Transfer is a cognitive reflex that is applied to all linguistic representations that are send to the syntax-semantics interface for interpretation. 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.

At any given moment during a linguistic conversation or communication the hearer maintains a transitory repository of semantic objects called *global discourse inventory* that the conversation "is about." Thus, if we talk about a person called John, the discourse inventory will contain a representation of John. The objects maintained in the discourse inventory are language-external

[11] Transfer constitutes a "reverse-engineered chain creation algorithm" within the context of modern generative grammar.

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 induce 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 viewed as 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. This conversion happens inside *narrow semantics*. We can therefore conceptualize narrow semantics as a structure that mediates communication between the language faculty, the syntactic pathway more specifically, and the language-external systems that constitute global cognition and access the discourse inventory of transient semantic objects activated during the conversation or communication. A *conversation* is defined as a sequence of sentences that share the global discourse inventory, making it possible to use introduction sentences for populating the inventory and test sentences that make claims about those entities (e.g., *John*$_1$ *admires Mary*$_2$; *he*$_1$ *likes her*$_2$).

3.4 Enumerative, recognition and generative grammars

Sometimes an approach of such as the present one is criticized or dismissed on the grounds that it is perceived as irrelevant for (generative, autonomous) linguistics, being allegedly concerned with performance, comprehension or other aspects of linguistic processing and data that is viewed as less relevant for linguistic theorizing.

The framework elucidated in § 3.4 represents a *recognition grammar* in the sense that it maps input sentences into grammaticality judgments, derivations, syntactic analyses and semantic interpretations. The alternative is an *enumerative grammar* which produces sentences instead of recognizing them. It is important to realize that both recognition grammars and enumerative

grammars are generative grammars: they define sets of grammatical expressions.[12] The former defines the set by providing that set's characteristic function, the latter by enumerating the members of the set. It is thus possible to transform a recognition grammar into an enumerative one for example by creating a function which feeds the former with all possible word order combinations given lexical selection and outputs the accepted inputs that then constitute the set of grammatical sentences targeted for study. In this way it is possible to use any recognition grammar, including the syntactic engine of the present mode, as an enumerative model. Similarly, an enumerative model can be transformed into a recognition model by verifying whether any given input sentence is in the set produced by the enumeration. Thus, the two models are mathematically equivalent under very weak assumptions (e.g., the computations must be effective).[13] This means that most generative grammars can be written without losing empirical coverage either as recognition grammars or as enumerative grammars. In addition, it is possible to take an enumerative grammar and reverse-engineer it into a recognition grammar, and vice versa. Reverse-engineering may improve or lose empirical attractiveness (elegance, simplicity, naturalness), so the choice is not entirely free.

The framework described in § 3.1 begins from the point of view of language comprehension, thus the input sentences are linear lists of phonological words. The derivation then proceeds from left to right and uses a separate presyntactic lexico-morphological component for decomposing words into their grammatically relevant constituents. This is, indeed, the intended interpretation of the model. This does not imply that the model is *restricted* to language comprehension, however. It is possible to imagine a realistic language production model which creates utterances by

[12] The term "generative grammar" means a grammar that defines an explicit set of (grammatical) expressions in one or several languages. Therefore, recognition grammars and enumerative grammars are generative grammars by definition. On the other hand, most sketches of "generative grammar" proposed in the literature today are not generative grammars in the literal sense, as they do not generate anything.
[13] This does not mean, however, that the two approaches are empirically irrelevant notational variations. For example, the recognition grammar assumed in the present work relies on linear order when it creates the set of possible derivations. Many enumerative grammars available today do not use linear order in the input; rather, the input is a set of lexical items. There are other differences between the approaches, some minor, others major.

proceeding from the left to right, essentially producing the utterance in the order they are spoken. It is well within the realms of empirical possibility. Such model would likely not use phonological words as starting point, but abstract lexical items and then employ "free will" for syntactic branching. The former option is currently available, thus the user can use a string of abstract lexical items in the input string instead of phonological words, but the latter is not.[14]

Given that enumerative and recognition grammars are in many ways equivalent, are there any grounds for selection one or the other? The differences are empirical and thus depend ultimately on the nature of the data, not a priori concerns. The matter boils down to the question of what the cognitive principles of language are. In a comprehension-based recognition grammar such as the present one the most fundamental grammatical principles are mechanisms which map sensory inputs into abstract representations, whereas in the typical production-based enumerative grammar they are principles of bottom-up derivations which create linguistic representations near or at the interface between syntax and semantics and the transform them for production and spellout. For example, while in the former the derivation *begins* from linearized inputs, in the latter linearization takes place at the *end* of the pipeline. Another difference concerns practical work. An enumerative grammar, as the name implies, outputs a set of expressions whose correctness the researcher must verify. Thus, each input is mapped into a set. A recognition grammar processes sets of expressions or perhaps just one expression. The former therefore is much more resource-intensive than then latter.

3.5 Linear phase model and other theories of grammar

The linear phase algorithm instantiates a generative grammar in the sense of § 3.4. It is designed to be an observationally, descriptively and explanatorily adequate model. For these reasons alone, it can be viewed as representing a class of generative grammars. Indeed much of the terminology

[14] Modeling "free will" does not constitute at present a meaningful scientific problem. The production model should produce all sequences of words possible, given the initial string of lexical items.

and theoretical assumptions come from recent generative theorizing, minimalism in particular (Chomsky, 2000, 2001, 2004, 2008). The left right assembly was originally proposed by (Phillips, 1996), while much of the overall orientation though not the specifics owes to the dynamic syntax framework (Cann et al., 2005a, 2005b; Kempson et al., 2001). Another approach that has influenced the current model are the top-down grammars (Bianchi & Chesi, 2010, 2014; Chesi, 2004, 2012, 2013; Zwart, 2009, 2015).

4 The core syntactic engine

4.1 Introduction

This section provides the minimal specifications that makes it possible to build up the kernel of the model (i.e. its syntactic engine) from scratch. The text is written for somebody working with a concrete algorithm or wants to understand the logic of the source code. Empirical matters dealing with the linguistic theory are discussed in the published literature and are omitted here.

4.2 Merge-1

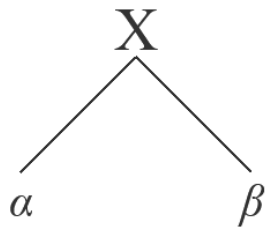
Linguistic input is received by the hearer in the form of sensory stimulus. We can first think of the input, to simplify the situation to bare essentials, as a one-dimensional string $\alpha * \beta * \ldots * \gamma$ of phonological words. In order to understand what the sentence means, the human parser (which is part of the human language faculty) must create a sequence of abstract syntactic interpretations for the input string received through the sensory systems. One fundamental concern is to recover the hierarchical relations between words. Let us assume that while the words are consumed from the input, the core recursive operation in language, Merge, arranges them into a hierarchical representation. 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. The same operation can be described in an image format as follows:



Here X denotes the complex constituent [α, β], α is the left constituent, β the right constituent. 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 they are represented as being part of the same linguistic chunk. We can attend to both of them as one object, manipulate them as part of the same representation, and in general perform operations that takes them both into account. Therefore, 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 combines phonological words. While this is a possibility, it would not be linguistically useful assumption. We assume that (3) is mediated by *lexicon*: a storage of linguistic information that is activated on the basis of the original phonological words. The lexicon maps phonological words in the input into *lexical items* which contain *lexical features* § 4.9.4 such as lexical categories (noun, verb), inflectional features ('third person singular') and meaning ('John', 'sleeping')(for a list of lexical features. The lexicon and lexical retrieval are handled inside its own module, discussed in § 4.9. I will assume from this point on that syntax is operating with lexical items, not with phonological words.[15] It is possible, however, to omit the phonology-lexical mapping in its entirety and feed the model directly with the lexical items.

[15] Whether phonological information flows into syntax is an interesting empirical question. The current algorithm allows some phonological information to pass into further processing stages, but the empirical data motivating this assumption is quite peripheral.

The resulting complex chunk [α, β] 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 consume the sensory input from left to right, (3) implies that the constituent that arrives first will be the left constituent.[16] Thus, we can think of the left constituent as the "first constituent." Because the configuration is asymmetric, it can be viewed as a list that can contain other lists as constituents.

The theory itself places no limited on the number of elements that can be assembled together, although there are several other cognitive bottlenecks which can limit the resulting structures and our ability to process them. Suppose the next word is *furiously*. Example (4) shows three possible attachment sites, 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 assemble the structure from left to right. 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

[16] The left to right assembly mechanism was proposed originally by (Phillips, 1996).

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.[17]

This leaves (4)(b) and (c). The parser will select one of them. Which one? There are many situations in which the correct choice is not known or can be known but is unknown at the point when the word is consumed. An incremental parsing process must nevertheless make a decision. Let us assume that all legitimate merge sites are ordered and that these orderings generate a recursive search space that the algorithm will explore by backtracking. 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

[17] These assumptions depend of course on prior assumptions concerning linearization. The present model assumes an exceptionless left-to-right depth-first algorithm. Suppose the researcher assumes that linearization is based on some alternative rule L; then Merge-1 must consider different alternatives based on which configurations could have in principle produced the input string relative to L.

(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 the ranking.

The 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 (6)b involves extra effort when compared to (a). At the point where the incremental parser encounters the last for *fell*, it has (under normal language use context) 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 discovers a solution in which *raced* is interpreted as a noun-internal relative construction [$_{DP}$ *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-1, the two solutions *raced* = finite verb and *raced* = participle are ranked so that the former is tried first.

The backtracking search can be interpreted in one of two ways. If looked from the point of view of the recognition grammar, putting questions of performance aside, its purpose is to consider all possible derivations available given the input and some grammar. That is, the procedure will find all attested ambiguities from the input and rule out unattested syntactic and semantic analyses. Had we limited the search only to the first-pass parsing solution most of the derivations latently present in our grammar would not have been tested for correctness. Once we have an observationally adequate model which finds all attested ambiguities from the input and only those, we can interpret the search as a psycholinguistic process and try to map it with behavioral

evidence acquired from psycholinguistic experiments. This is, however, optional. In addition, it is not necessarily true that speakers use full recursive backtracking in real language comprehension. If they do not use recursive backtracking, then that operation must be interpreted as a calculation device or testing procedure while additional mechanisms model realistic language comprehension.[18] Aspects of realistic language comprehension are discussed in § 5.

Merge-1 can break constituency relations established at 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 no longer holds: *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 of this is that if we merge two words as sisters, we do not 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 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*. Only those configurations

[18] My own working hypothesis is that realistic language comprehension does not rely on recursive backtracking but processes the sentence anew from the beginning while adding random noise to the decision making in an attempt to find alternative parsing paths.

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.

4.3 Lexical selection features

Consider a transitive clause such as *John admires Mary* and how it might be derived under the framework 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. If we do not determine it, then sentences such as **John admires to want him* or **John admires admires Mary* will be interpreted as grammatical. 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. When *Mary* is consumed, the operation checks if any given merge site allows/expects the operation. In the example (10), the test passes: the label of the selecting item matches with the label of the new word.

(10) John admires Mary

↓ ↓ ↓

[John [admires Mary]]

[!COMP:D] [D]

Feature [COMP:L] means that the lexical item *licenses* a complement with type [L], and [!COMP:L] says that it *requires* a complement of the type [L]. Feature [−COMP:L] says that the lexical item does *not* allow for a complement with label L.[19] When the parser is trying to sort out an input, it uses lexical features (among other factors) to rank solutions. When the phrase structure has been completed, and there is no longer any input to be consumed, these features can be used for filtering. Filtering is performed at the LF interface (discussed later) and will be called the *LF legibility test*. It checks if the solution provided by the parser makes sense from a semantic point of view. Sentence **John admires to want him* will be judged ungrammatical because *admire* cannot select infinitival complement clauses.

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 this 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 prefers, is allowed or required to have. I call such features *specifier selection features*. A feature [SPEC:S] ("select the whole clause") favors solution (a), [SPEC:V] favors solution (b)("select the verb phrase"). What constitutes a specifier selection feature will guide the

[19] The behavior of [!COMP:D] and [−COMP:D] is intuitively obvious, but there is redundancy between licensing and when there is no feature at all, as in the latter case the complement is neither mandated nor rejected, hence it could also be considered 'licensed'. Licensing is currently used in the parsing process and also in some grammatical roles such as in distinguishing thematic heads from nonthematic ones; whether it could be eliminated in its entirety from the theory is an interesting open question.

selection of a possible left sister during comprehension.[20] Most of the lexical features are enumerated in § 4.9.4.

4.4 Phases and left branches

Let's 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*: the derivation can forget them inside that particular parsing path.[21] It follows that if no future operation is able to affect a left branch inside the that parsing path, 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, the original merge operation that created the left branch phase can 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. Since the model can backtrack, it is able to reconsider all left branches. We can formulate the condition tentatively as (13).

[20] Because constituency relations may change at some later stage of the derivation, we do not know which pairs will constitute 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.

[21] They are phases in the sense of (Chomsky, 2000, 2001), except that the units differ due to the reverse-engineered architecture. See (Brattico & Chesi, 2020).

(13) *Left branch phase condition*

Derive each left branch independently.

What we mean by deriving something "independently" becomes unambiguous only when we have a fully described grammar, but intuitively it says that if there are syntactic operations that must be done to the left branch before it is transferred to semantic interpretation and/or before we can consider further merge-1 operations, then such operations must be done at the point when the left branch is created. This is the intended interpretation of (13), although it leaves it open what these "operations" are.

4.5 Labeling

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

(14) Mary admires John's mother

↓ ↓ ↓ ↓

Mary [admires [John's mother]]

Now recall that the verb's complement selection feature [!COMP:D] refers to the label [D] of the complement. What is the relationship between [D] and the phrase *John's mother* that occurs in the complement position of the verb in the above example? Since any constituent may contain an arbitrary number of elements, the mapping from complex constituents into labels like [D] is not trivial. How do we know, from any arbitrarily complex constituent, what is type is? The answer is provided by (15).

(15) *Labeling*

Suppose α 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 starting from the top. "Closest" means closest form the point of view of the selector. If we analyzed the situation from the point of view of the labeled phrase itself, then closest is the highest or most dominant head. Principle (15) means that labeling ignores right adjuncts (this will be in fact a defining feature of adjuncts). Example (16) illustrates some basic examples.

(16) a. [$_{VP}$ *admires Mary*] = left primitive verb;

b. [$_{PP}$ *from Mary*] = left primitive preposition;

c. [$_{VP}$ [$_{DP}$ *that man*][$_{VP}$ *admires Mary*]] = verb is the first left primitive;

d. [$_{DP}$ *the man*] = definite determiner is left, hence the label;

e. [$_{VP}$[$_{VP}$ *admires Mary*]⟨*furisously*⟩] = adjunct is ignored;

f. [$_{NP}$ [$_{DP}$ *that man's*] *car*] = right primitive noun.

Notice that complex constituents [α β] themselves do not have labels or type; its label is determined by algorithm (15). The rule is defined in this way because this definition has generates correct and/or plausible results; the matter is empirical, however, and it is easy to experiment with different alternatives.[22] Since the phrase structure geometry can change during incremental parsing, also labeling may change. This is shown by the following derivation:

(17) a. [$_{DP}$ *the man*] + admires =

b. [$_{VP}$ [$_{DP}$ *the man*] *admires*].

What was originally a DP was transformed into a VP after the verb was merged. This is an important feature of parsing, because it means that we do not have to fit the incoming words into

[22] For example, these assumptions rule out configuration [$_{βP}$ $α^0$ β] where a primitive left constituent does not project. We cannot therefore have a situation where a head moves to the specifier position of another head, in the terminology of standard linguistic theory. We can, however, contemplate modifications to the labeling algorithm such that this becomes a possibility.

existing or presupposed grammatical templates; the template is created dynamically as more words as assembled together.

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*.[23] Consider again the derivation of (3), repeated here as (18).

(18) John + sleeps.

↓ ↓

[John, sleeps]

If *John* is a primitive constituent, labeling will provide [$_{NP}$ *John sleeps*] which means that the sentence is interpreted as a noun phrase. However, (18) is a sentence or verb phrase, not a noun phrase. For example, it cannot be selected by transitive verbs (**John admires John sleeps*). We therefore assume that *John* is a complex constituent with the structure [$_{DP}$ D N]. This information comes from the lexicon, where proper names are decomposed into D + N structures. The structure of (18) is therefore (19).

(19) John + sleeps.

↓ ↓

D N sleeps

↓ ↓ ↓

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

The structure of the lexicon is examined in detail in § 4.9.2, at this point it is important to note that we have nowhere assumed that phonological words are mapped into primitive lexical items. While some words are, others aren't. The point is, however, that whether some word is or isn't

[23] A constituent that has only the left or right constituent, but not both, will be primitive according to this definition.

decomposed is crucial for labeling, because the labeling algorithm is sensitive to the distinction between primitive and complex constituent.

4.6 Upward paths and memory scanning

Constituents making up phrase structure representations can enter into several types of dependency relationships. Consider the following pair of Finnish sentences.

(20) a. Pekka ei ihaillut *Merja-n/ Merja-a.
Pekka not admire Merja-acc Merja-par
'Pekka did not admire Merja.'

b. Pekka on ihaillut Merja-n/ *Merja-a.
Pekka has admired Merja-acc Merja-par
'Pekka did admire Merja.'

The case form of the direct object argument depends on polarity: negative clauses require that the object is marked by the partitive case, glossed as PAR, while affirmative clauses require the accusative case, glossed as ACC. Both sentences, when generated by merge-1 from the input, generate a representation approximated in (21). The dependency between the negation and the direct object is shown by the line under the sentence.

(21) Pekka ei/on ihaillut Merja-a.
[Pekka [not/is [admire Merja]]]
└──────────┘

This relationship cannot depend on local selection described in § 4.3. Dependencies of this type are modelled by relying on the notion of *path*. The case form at the case assignee must enter into a "compatibility check" with another element, the case assigner. In this case the case forms accusative and partitive enter into a compatibility check with the polarity element. It does this by forming an *upward path* from the case assignee to the case assigner. Suppose α is an element requiring checking; then

(22) *Upward path*

the upward path from α contains all constituents that dominate α plus their immediate daughters.

For example, if the phrase structure is (23)

(23) $[_{AP} \ldots A [_{BP} \ldots B [_{\alpha P} \ldots \alpha \ldots]]]$

then the upward path from α contains αP, BP and AP (the dominating constituents) plus their immediate constituents A and B. The dependency itself is formed by scanning through the path to a target element, and the first target element available is always selected. If the target is not found, the operation fails. This mechanism is applied to case checking, binding, adjunct positioning, control and operator scope calculations among many other nonlocal grammatical operations. The path coincides with or perhaps defines the content of the "syntactic working memory" when α is processed.[24] The path mechanism was originally inspired by the connectness approach of (Kayne, 1983, 1984).

4.7 Adjunct attachment

Adjuncts, such as adverbials and adjunct PPs, present a challenge for any incremental parser. Consider the data in (24).

(24) (Finnish)

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

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

[24] Recall that all left branches are phases, thus they have already been processed and moved out of the active syntactic working memory when α is targeted.

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 almost completely unpredictable.

Adjuncts are assumed to be geometrical constituents of the phrase structure but they are stored inside a ‘parallel’ syntactic working memory making them invisible for sisterhood, labeling and selection in the primary working memory. They increase the dimensionality of the phrase structure. The idea is illustrated in Figure 24.

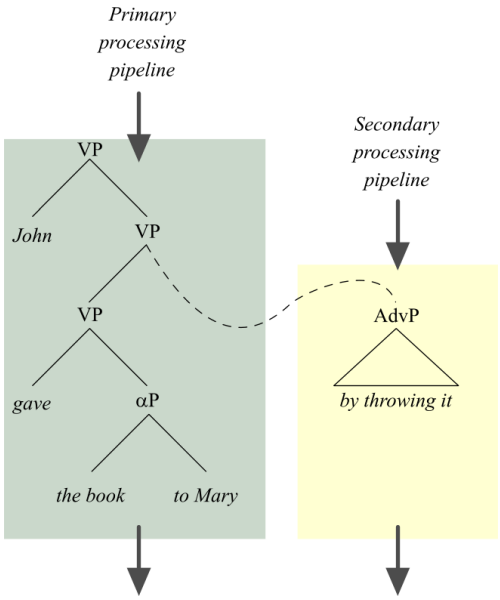


Figure 24. Adjuncts are geometrical constituents that are pulled out of the primary working memory and are processed inside separate processing pipeline. We can imagine that the VP is made up of two segments.

Thus, the labeling algorithm, as specified in § 4.5, ignores adjuncts. The label of (25) 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.

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

Consider (26) next.

(26) 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 sisterhood in such a way that 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 the complement position.[25] 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 from these assumptions, however, that adjuncts could be merged anywhere, which is not correct. Each adverbial (head) is associated with a feature linking it with a feature or set of features in its host structure. In this case the linking relation is established by a *tail-head dependency*: we imagine that the adjunct, which exists in the separate syntactic plane, constitutes a "hanging tail" related to the head. 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 (ii) can be linked

[25] From the point of view of labeling, selection and sisterhood, the structure is therefore $[_{VP}[_{DP}$ *John*$]$ *sleeps*].

with the corresponding head by means of an upward path. Conditions (i-ii) have slightly different content and are applied in different circumstances.[26]

(27) Condition on tail-head dependencies

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

a. α occurs inside a projection whose head β_F, or HP is β_F's sister;

b. α can be linked with β_F through an upward path (22).

B_F denotes a head (or phrase) that has feature [F]. Condition (27)(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 a feature. It does not restrict the position at which an adjunct must occur inside αP. Condition (27)(b) allows some adjuncts, such as preposition phrases, remain in a low right-adjoined or in "extraposed" positions in a canonical structure. 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 § 4.8.6.

The definition of adjunction licenses both left and right adjuncts; what matters is that the adjunct is ignored by labeling and sisterhood. Left adjuncts are very similar to regular left phrases, however, because left phrases, whether adjuncts or not, are ignored by the labeling algorithm. Notice that the label of [α β] will be β if α is complex, it does not have to be an adjunct; if it is an adjunct, the situation will be the same. Left phrases and left adjuncts are not identical, of course, since the former can be selected while the latter can't; they generate different sisterhood dependencies. They differ also behaviorally. One differences is that left adjuncts can be stacked while left adjuncts cannot.

The definitions leave room for a situation where an adjunct phrase contains another adjunct phrase. We would then have the primary structure with a path into the secondary structure, and

[26] This is an imperfection in the model that needs resolving.

another path into the tertiary structure. This is of course perfectly normal in language where an adverbial phrase, for example, may contain further modifiers. There is no in principle limit on the number of adjunction dimensions any given structure can have, although they could become difficult to process.

Adjunct processing as a whole is a complex topic, and the present version (15.0) still awaits several simplifications to be implemented and tested.

4.8 Transfer

4.8.1 Introduction

After the spellout structure has been composed (more exactly, a phase), it will be *transferred* to the syntax-semantics interface (LF interface) for evaluation and interpretation. Transfer performs a number of noise tolerance operations removing 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.

4.8.2 Phrasal reconstruction: general comments

A phrase or word can occur in a *canonical* or *noncanonical* position. A canonical position in the input could be defined as a position such that given the regular Merge-1 operations the constituent ends up in a position where 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 (28) illustrates a sentence where this is the case.

(28) John admires Mary

↓ ↓ ↓

[$_{VP}$John [$_{VP}$ admires Mary]]

Legitimate output is reached by merging-1 the input words into the phrase structure: the operation will bring all arguments inside the verb phrase where their correct thematic roles are determined. Example (29) shows a variation in Finnish where this is no longer the case.

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

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

(30) [who [Pekka admires]]

This solution violates two selection rules. There are two specifiers at the left edge, *who* and *Pekka*, and *admire* lacks a complement. Furthermore, even if the parser backtracks, it cannot find a legitimate solution. A candidate structure such as [$_{VP}$ [$_{DP}$ *who Pekka*] *admires*] is also illegitimate.[27] The two violations are not independent of each other, however: the element that triggers the double specifier violation at the left edge is the same element that is missing at the complement position. This is clearly not a coincidence: the interrogative pronoun causes these problems because it has been "dislocated" to the noncanonical position from its canonical complement position. We can see this by consider what would fix both problems: movement of the interrogative pronoun from the left edge to the complement position of the verb, thus something like *who Pekka admires* → *Pekka admires who*.

Dislocation is a persistent and general feature of all natural languages. In Finnish, for example, almost all word order variations of the referential arguments of a finite clause are possible.

[27] In principle we could feed (30) directly to the LF interface component. This representation 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 would accept **John admires* or even **John admires Mary to leave.* Similarly, the main verb is here preceded by two argument DPs, but this too can produce ungrammatical results such as **John Mary admires*.

Application of the straightforward Merge-1 would result in a situation where each word order is represented by a different structure that do not necessary share any properties, yet most of these word orders create stylistic differences. For example, the two orders in (31) correspond to the same core proposition ‘Pekka admires Merja’.

(31) a. Pekka ihailee Merja-a. (SVO)
Pekka.nom admires Merja-par
’Pekka admires Merja.’

b. Merja-a ihailee Pekka. (OVS)
Merja-par admires Pekka.nom
’Pekka admires Merja.’

Unless the system perform noise tolerance normalization where all selection restrictions can be checked, we cannot filter out ungrammatical sentences from the grammatical ones. It is clear, furthermore, that Finnish speakers use overt case forms, here the nominative and partitive, to do this. The model must therefore “reverse-engineer” the construction in order to create a representation that can be interpreted at the LF interface. *These operations take place during transfer*.

Almost all reconstruction operations are based on the same computational template. First the system recognizes that the element occurs in an illegitimate position where it cannot satisfy one or several LF interface condition(s), such as selection or case checking. Once the system detects the presence of an offending constituent, it will attempt reconstruction, where a legitimate position is sought. If a legitimate position is found, the element is copied there, and the search ends; if a legitimate position is not found, the constituent remains in its surface position. In that case, some later operation may still dislocate it. How much variation is allowed in any given language depends on the properties of reconstruction and transfer: it is possible that transfer fails to recover the expression, resulting in deviance or ungrammaticality.

4.8.3 Ā-chains

Let us return to the simple interrogative clause cited earlier, repeated in (32).

(32) Ketä Pekka ihailee?
who.par Pekka.nom admires
[$_{VP}$ who [$_{VP}$ Pekka admires]]

Let us assume (to simplify) that Merge-1 creates the structure shown on the third line. The verb has two specifiers and no complement. One of these errors triggers reconstruction. In the current algorithm, it is triggered by the presence of an extra specifier *ketä* 'who.par'. The reconstruction algorithm searches for the first gap for this element where it can generate a legitimate position and discovers (33).

(33) [~~who~~ [Pekka [admires who]]]
└──────────────────┘

This representation, which is created during transfer, is transferred to the LF-interface. The interrogative pronoun is copied from SpecVP into CompVP, and will be interpreted as the patient of the verb *admire*. Both selection violates have now been fixed: there are no longer two specifiers at the left end of the sentence, and the complement is no longer missing.

The search algorithm proceeds from the deviant element downstream until either a suitable position is found or there is no more structure. It cannot go upwards or sidewards. 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* (§ 4.8.7). 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) it does not violate any other condition for LF objects. If no such position is found, the element remains in the original position and may be targeted by 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.

This normalization operation is not yet sufficient. One problem is that the original sentence represents an interrogative clause – the speaker is asking rather than only stating something – that can only be selected by certain verbs. For example, while it is possible to say *John asked who John admires*, it is not possible to say **John claimed who John admires*. The bare VP representation does not yet represent this fact, because it was always interpreted (implicitly, as it were) as a declarative proposition. Interrogativization is represented by a *wh*-feature that must be part of the highest head in the sentence, so that it can be selected by *ask*/*claim*. This is accomplished by copying the *wh*-feature from the fronted interrogative pronoun to the local head (34)(a) or, if no local head is available, it is generated to the structure and then equipped with the *wh*-feature (b).[28]

(34) a. Who admires Pekka?

[$_{VP}$ who$_{wh}$ [$_{VP}$ admires$_{wh}$ Pekka]]

b. who Pekka admires?

~~who~~$_{wh}$ C$_{wh}$ [Pekka [admires who$_{wh}$]]

C$_{wh}$ represents the extra head generated to the structure to carry the *wh*-feature. Notice that in both cases, a higher verb can now select the interrogative feature irrespective of whether it is inside the verb or the C-element (35).

(35) a. John asks/claims + [$_{VP+wh}$ who [admires$_{V+wh}$ Pekka]

b. John asks/claims + [$_{CP+wh}$ ~~who~~ C$_{wh}$ [Pekka [admires who]]]

The reconstruction algorithm generates the C head into the structure. We can think of this operation as reconstructing a phonologically null head, i.e. a head that had no direct representation in the sensorimotoric input.

[28] Feature copying replaces feature checking of the enumerative generative grammar.

The dislocated phrase also 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'). Let us consider how dislocation represents the propositional scope of an interrogative clause. First, the reconstructed interrogative pronoun or phrase is called an *operator*. This means that an argument such as 'who' does not refer to anything by itself; it is an "argument placeholder." We can also think of it as a variable, the target of the question that must be filled in by the "answer." Then, any finite element that contains the same operator feature, in this case *wh*-feature, will determine the scope.

(36) ~~who$_{wh}$~~ C$_{wh,fin}$ does John admire who$_{wh}$

The operator-scope dependency, marked by the line in the representation above, is checked by an operator-variable module inside narrow semantics, which pairs the operator with its scope marker. If feature [OP] (e.g. *wh*) represents the operator, then the closest head with [FIN][OP] will be the scope marker. Together they create the Fregean interpretation 'which x: John admires x?'. These dependencies, which have both a syntactic and semantic dimension, are called *Ā-dependencies* or *Ā-chains* in the literature. Chains have a syntactic side, because the dislocated deviant element must be reconstructed by a syntactic operation that takes place during transfer, by minimal search; and a semantic side, because the operator must be linked with the corresponding scope marked inside the semantic system. Both operations can fail, and when one or both fail, the input sentence is judged ungrammatical.[29],[30]

[29] This architecture was originally inspired by Chomsky's dual interpretation model (Chomsky, 2008). Operator features involved in these mechanisms are interpreted by a special semantic system (the operator-variable module inside narrow semantics).

[30] There is some variation with respect to how operator-variable constructions such as interrogatives are packaged for communication. In some languages, the operator remains in its canonical position and is not fronted at all to the beginning of the clause; then there are languages where several operators can be fronted. Thus, the fact that both Finnish and English front interrogative pronouns is not inevitable or necessary.

Traditional enumerative grammars describe the same process – Ā-chains – by moving the interrogative pronoun from the complement position into the left edge position where it represents the scope of the question. The operation is called Move or Internal Merge. Reconstruction is its mirror image, a reverse-engineered Move operation; but it is important to realize that they describe the same object, a directionless chain that is constituted by the two (or more) members in two (or more) syntactic positions.[31]

4.8.4 A-chains

In addition to Ā-chains, discussed in the previous section, many languages exhibit another type of phrasal reconstruction, creating A-chains. To see how, consider that English preverbal subject position can be occupied by both the agent and patient (37).

(37) a. John admires Mary.
[John [admires Mary]]

b. Mary was admired (by John).
[Mary [was [admired (by John)]]]

The direct Merge-1 solution (second lines in the above example) is wrong in the case of (b), because *Mary* would be in the agent position, *John* the patient (if present). However, we can conclude on the basis of data of this type (a–b) that the English preverbal subject position is *not* a thematic position. It can be occupied by both agents (a) and patients (b). This is supported further by the data from Finnish, which shows that argument order does not correlate necessarily with thematic interpretation. This brings us to the presence of the tensed auxiliary verb *was* in the example (b). Note that both the bare finite verb (a) and the aux-verb construction (b) involve a

[31] Whether chains are visible and, if they are, whether there is an operation that is sensitive to such objects are interesting nontrivial empirical questions. What we mean by using the term "chain" here is that that there is an operation (Move, Reconstruction) that brings them into existence. The operation linking variables to their scopes is not sensitive to the chain as such, but is based on lexical features + upward pathing and reads information created by reconstruction. However, there have been several versions of the algorithm in which chains were considered real objects on their own right when required by empirical evidence.

tensed finite verb; when the sentence contains an auxiliary, tense information is expressed by the auxiliary. Let us assume that tense represents an independent packet of grammatical information in the sentence that may be expressed either by combining it with the verb (a) or by carrying it by an otherwise dummy auxiliary verb (b). We can depict the situation by using the schema (38).

(38) [$_{TP}$ John [$_{TP}$ T [$_{VP}$ admire Mary.]]]

Tense T can be expressed either by an auxiliary (*was, does*) or by combining it with the main verb; the latter operation will be discussed later. Recall that we already assumed in §4.5 that phonological words such as *John* decompose into several lexical items D and N; now we apply this scheme to tensed verbs, and assume that they are decomposed into T and V. Having assuming this, we can express the intuition that the preverbal subject position is not a thematic position and that the thematic roles are assigned inside the VP: the preverbal subject position is SpecTP and the thematic agent position SpecVP. Then we can assume that the grammatical subject is reconstructed from the former into the latter (39).

(39) [$_{TP}$ ~~John~~$_1$ [$_{TP}$ T [$_{VP}$ John$_1$ [$_{VP}$ admire Mary.]]]

This operation called *A-reconstruction* and it forms an *A-chain*. The operation is triggered by the fact that SpecTP is not a thematic position – hence the referential argument cannot remain at this position at the LF-interface – and the operation locates the first position where it can be interpreted. The fact that SpecTP is a nonthematic position is marked in the current implementation by feature [EF]. A-reconstruction differs from Ā-reconstruction in that the former does not form operator-variable constructions inside the semantic system and is not triggered by operator features such as *wh*-features.[32]

[32] The EPP no longer exists as a monolithic lexical feature but is replaced by a function EF that relies on a set of edge features. When these details do not matter I will keep using the term "EPP."

Representation (39) is very close to how the model represents canonical transitive or intransitive clauses. We assume that they are composed out of at least two separate lexical items T and V, which together form a *complex predicate*. Thematic roles are assigned inside the VP, while T is an extra part of the complex predicate that does not assign a new thematic role but modifies the predicate itself. That is, the sentence means 'John T + V Mary', where T and V are part of the same predicate. There is only one detail that is still missing, namely the difference between transitive and intransitive verbs.

4.8.5 Head reconstruction

Let us consider again (40).

(40) John admires Mary.

So far we have assumed that the tensed finite verb is made of at least two grammatical heads or independent packages of grammatical information – tense T and the verb stem V. The consequence is that the sentence has one extra position that is nonthematic, namely the preverbal subject position SpecTP. We noted that the T can be expressed either by an independent auxiliary element (*does*, *was*) or by the finite verb, but what was left unexplained was how the finite verb is dissolved into the two independent components when that strategy is used. *Head reconstruction* solves this issue.

First we should note that whether and how grammatical heads are chunked into morphological and phonological words is to some extent arbitrary and subject to crosslinguistic variation. Thus in English, we can express the same tense information by *John does admire Mary* and *John admires Mary*, where in the latter the two heads have been chunked together. Head reconstruction must therefore contain an operation that recovers morphological chunks from phonological words. This part is handled by the lexico-morphological module. It takes phonological words as input and decomposes them into their constituent morphemes, thus *admires* ~ T + V. These heads are forwarded into the syntactic component and merged-1 to the syntactic structure.

In the first stage, a *complex head* $T(V)^0$ is created inside the syntactic component. Thus, the lexical streaming operation maps lexical decompositions into complex heads: *admires* ~ T + V ~ $T(V)^0$. *Head reconstruction* extracts the heads and distributes them into the structure, $T(V)^0$ ~ [T … [...V…]]. The steps are illustrated in (41).

(41) John admires Mary. (Input)

[John $[T(V)^0$ Mary]] (Output of Merge-1)

[John [T [V Mary]]] (Output of head reconstruction)

When V is extracted out from T, the closest possible reconstructed position is searched for and the element is merged into that position. The configuration shown in (41) is selected because T can select a VP complement.

We assumed earlier that the phrase structure is made up of primitive constituents, which are elements that have zero daughters, and complex constituents, which have two daughters. Thus, if α and β are primitive constituents, then $[\alpha\ \beta]$ is a legitimate complex constituent. Because $[\alpha\ \beta]$ will always be treated like a phrase, we can also say that it is a *phrasal constituent*. What we have not considered is a situation where the complex constituent has only one daughter constituent (left or right). This configuration creates complex constituents that are nonphrasal, which, then, are the complex heads referred to above. Thus we have $\alpha(\beta)^0 = [_\alpha\ \beta]$, where α is a constituent that has only one (right) constituent β. The mechanism is iterative, and allows the system to chain several heads into ever more complicated words, for example $A(B(C)))^0 = [_A\ [_B\ C]]$. Phrasal syntactic operations do not treat nonphrasal complex constituents (complex heads) as phrases (observing the lexical integrity principle), while they can still contain several grammatical heads (primitive lexical items) ordered into a linear sequence.[33]

[33] Morphologically complex words are linear sequences of (sensorimotoric) elements. We can think of the lopsided constituency relation as representing simple "followed by" relation: produce α followed by β, in short α * β. Each element in the sequence must be atomic and map into one sensorimotoric program.

These assumptions raise the question of why we compose complex heads at all instead of mapping the different parts of words directly into the syntactic structure. That is, if the lexico-morphological component can decompose admires into T + V, why not the take these lexical items and merge-1 them directly in [...[T...[...V...]]]? In fact, earlier versions of the model did exactly that. The reason this was rejected later is because there is empirical evidence suggesting that grammar can create and manipulate complex heads. For example, in Finnish it is possible to move complex predicates as shown in (42).

(42) Myy-dä-kö$_1$ Pekka halusi __$_1$ koko omaisuutensa?
sell-a/inf-q Pekka wanted all possessions
'Was it selling that Pekka wanted to do with all his possessions?'

The current model handles constructions like (42) exhibit long head movement by reconstructing complex heads syntactically (Brattico, 2022a). If we assume, then, that syntax has access to complex heads, why can't we assume that such complex heads are already stored as complex heads in the lexicon? According to this alternative, the phonological word *admire* is mapped directly into T(V)^0, which is placed into syntax as a complex package. One reason why the algorithm does not work in this way is the existence of morphological parsing that is able to create and comprehend novel words that do not exist in the lexicon. This means that it must be possible for speakers to perceive words as composed out of individual morphemes, and this is what the lexico-morphological component accomplishes. In other words, there must exist a process which can decompose phonological words into parts that do not correspond to anything in the lexicon. But there is another reason complex heads are not stored in the lexicon, namely this would mean that lexical items are ultimately syntactic objects and, by the same token, that the component storing and manipulating them is part of syntax. In the current theory, the lexico-morphological component is extrasyntactic and only understands linear sequences of patterns; once we have syntactic objects, we are in the domain of syntax proper. The distinction between lexico-morphological processes ('non-syntax') and syntax is just made in this way. Positing a third

mediating structure, a syntactic lexicon, is possible but serves no purpose as far as the current evidence goes.[34]

4.8.6 Adjunct reconstruction (also scrambling reconstruction)

Thus far we have examined Ā-reconstruction, A-reconstruction and head reconstruction. There is a fourth reconstruction type called *adjunct reconstruction* (also adjunct floating, scrambling reconstruction). Consider the pair of expressions in (43) and their canonical derivations.

(43)

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 as being identical in these examples. The subject and object are in wrong positions. Neither Ā- 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 is very similar to the distribution of adverbials in Finnish, I have argued that richly case marked thematic arguments can be promoted into adjuncts (Brattico, 2020, 2022b). Case forms are morphological reflexes of tail-head features. If the condition is not

[34] There are grammatical framework and theories which posit complex syntactic lexicons in this exact sense. Whether they are better than the present model must be tested with real data in computational experiments. The question cannot be solved by a priori reasoning. Likewise, the hypothesis that we can do by mapping linear sequences of morphemes directly into syntactic structure, as assumed currently, is an empirical hypothesis.

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 the condition is satisfied. In this way, the inversed subject and object can find their ways to the canonical LF-positions (44).

(44) [⟨Merjaa⟩$_2$ T/fin [__$_1$ [käski [meidän [ihailla [__$_2$ ⟨Pekka⟩$_1$]]]]

Merja.par asked we.gen to.admire Pekka

'Pekka asked us to admire Merja.'

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

4.8.7 Minimal search

Let's consider again a standard interrogative sentences such as (45).

(45) Who does [$_A$ John's brother] admire __$_1$ [$_B$ every day]?

We have assumed that the interrogative pronoun is reconstructed into the canonical complement position marked by the gap __. The fronted interrogative and the gap form an Ā-chain. What has been left unsaid is how this reconstruction actually happens.

All reconstruction is based on *minimal search*. The gap position is located by descending downwards through the phrase structure from the position of the reconstructed element and detecting the first legitimate position (hence the term "minimal"). More formally, at each step the algorithm assumes a phrase structure γ as input and moves to α in $\gamma = \{_{\alpha P}\ \alpha\ \beta\}$ unless α is primitive, in which case β is targeted.[35] The operation therefore moves downwards on the phrase structure by following selection and labelling. Left and right (adjunct) phrases are ignored. The operation never branches since the algorithm provides a unique solution for any constituent. Consider (45). To reach the reconstruction site __$_1$, the search algorithm must avoid going into

[35] Notation $\{\alpha\ \beta\}$ refers to $[\alpha\ \beta]$ or $[\beta\ \alpha]$.

the subject A and into the temporal adverbial B. This is prevented by minimal search, which follows the projectional spine of the sentence and ignores both left specifiers/adjuncts and right adjuncts (there are no "right specifiers" in the model).

There is a possible deeper motivation for minimal search. Both left branches and right adjuncts are transferred independently as phases and are therefore no longer in the current syntactic working memory. The notion of minimal search coincides with the contents of the current syntactic working memory at any point in the derivation.

4.8.8 Agree-1

Most languages exhibit an agreement phenomenon, in which some element, such as finite verb, agrees with another element, typically the grammatical subject. This is illustrated in (46).

(46) a. John admire<u>s</u> Mary

b. *John admire Mary

c. *They admire<u>s</u> Mary

The third person features of the subject are cross-referenced in the finite verb. The term agreement or phi-agreement refers to a phenomenon where 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 (46).

A typical argument DP like *Mary* has interpretable and lexical phi-features which are connected to the manner it refers to something in the real or imagined world. Thus, *Mary* refers to a 'third person singular individual'. A predicate, on the other hand, must be linked with an argument that has interpretable phi-features. To model this asymmetry, a predicate will have *unvalued* phi-features, denoted by [φ_]. The value will be provided by the argument with which the predicate agrees with. This called valuation. The starting point is (47).

(47) Mary [admires John]

'argument' 'predicate'

D N T/fin

↓ ↓

[phi:num:sg] [phi:num:_]

[phi:per:3] [phi:per:_]

[phi:det:def] [phi:det:_]

The operation that fills the unvalued slots for the predicate by the phi-features of the argument is *Agree-1* (48).

(48) 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, the unvalued features disappear from the lexical item and the predicate is now linked with its argument. When agreement occurs between a head α and a phrasal argument, we call the operation *phrasal Agree*, abbreviated as $Agree_{LF}$.

Some predicates arrive to the language comprehension system with overt agreement information. The third person suffix +*s* signals the fact that the argument slot of *admires* should be (or is) linked with a third person argument. In many languages with sufficiently rich agreement, overt phrasal argument can be ignored. Example (49) comes from Italian.

(49) Adoro Luisa.

admire.1sg Luisa

'I admire Luisa.'

Thus, the inflectional phi-features are not ignored; instead, they are extracted from the input and embedded as morphosyntactic features to the corresponding lexical items, as shown in (50).

(50) John admire + s Mary.

↓ ↓ ↓

[John [admire Mary]]

[…3SG…]

This means that *admires* will have both unvalued phi-features and valued phi-features as it arrives to syntax from the sensory input and the lexico-morphological system. While this might be considered unintuitive, the former exists due to the fact that *admires* is lexically a predicate, while the latter are extracted from the input string as inflectional features (51).

(51) a. admire- = lexical predicate, hence it has unvalued phi-features [φ_];

b. -s = third person singular valued inflectional phi-features [φ].

c. = [φ_] + [φ].

A head may therefore also "agree with itself," an operation abbreviated as $Agree_{PF}$.

Existing valued phi-features impose two constraints 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. 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 subjectless pro-drop sentences. We thus interpret a valued phi-set inside a head as if it were a truncated pronominal element (52).

(52) adoro Luisa.

admire.1sg Luisa

admire.pro Luisa

'I admire Luisa.'

4.8.9 Ordering of operations

Ā/A-reconstruction and adjunct reconstruction presuppose head reconstruction, because heads and their lexical features guide Ā/A- and adjunct reconstruction. The former relies on EPP

features and empty positions, whereas the latter relies on the presence of functional heads. Furthermore, Ā/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 transfer sequence is therefore (53).

(53) *Ordering of operations during transfer*

a. Reconstruct heads →

b. Reconstruct adjuncts →

c. Reconstruct A/Ā-movement →

d. Agree-1 →

e. LF-interface and legibility →

f. Semantic interpretation.

The sequence is performed in a reflex-like manner. It is not possible to evaluate the operation only partially or to backtrack.

The sequence provided in (53) is compatible with two possible implementations, one where all operations are executed during one cycle and another where they form several cycles. In an ideal case, the algorithm would travel through the structure only once and fix all errors, in the order of (53), as they are encountered. This corresponds to the first option. This unification has not been accomplished in the present version due to the fact that the adjunct reconstruction algorithm (b) still differs too much from (a), (b) and (c). In addition, the transfer algorithm triggers separate extraposition operations that fixes certain issues with adjuncts. It is clear that the adjunct handling is not optimal and, once this issue has been sorted out, the whole sequence (53) could possibly be executed inside just one cycle.

4.9 Lexicon and morphology

4.9.1 From phonology to syntax

Most phonological words enter the system as polymorphemic units. They are made of stems, derivational bound morphemes, inflectional affixes, clitics, and prosodic information. The *lexico-morphological component* performs a mapping from input words into grammatically meaningful units by utilizing the *lexicon*. The lexicon is a list of *lexical entries* which are all key-value pairs where the key is a symbol that can occur in the input, and the value specifies the type of item it is mapped to (currently primitive lexical item, morphologically complex item, inflection or clitic). The lexicon is specified in ordinary text files the user can modify.

Let us consider the processing of morphologically complex words first. The lexicon matches phonological words with a *morphological decomposition*, if the latter is present. A morphological decomposition consists of a linear string of morphemes m_1#...#m_n. These morphemes, which (we assume) designate primitive lexical items, retrieve the corresponding primitive lexical items M_1 + … + M_n from the same lexicon which are fed into syntax, where they form complex heads $M_1(M_2…(M_n))^0$. The symbols M_1...M_n are keys in the same lexicon. We can imagine the morphologically complex words as "morphological chunks," lexicon information (e.g., M_1...M_n) that has been packaged into one unit. Head reconstruction (§ 4.8.5) extracts them into head spreads $[M_1…[M_2…[…M_n…]]]$. The whole process is illustrated in (54).

(54)	a.	John	admires	Mary.	(Input)
	b.	John	T+V	Mary	(Output of lexicon)
	c.	John	T(V)	Mary	(Input to syntax)
	d.	[John	[T [V	Mary]]]	(Output of head reconstruction)

Primitive lexical items are feature sets {f_1...f_n}. It is also possible to map a phonological word directly into a primitive lexical item. For example, the English definite article *the* is currently modelled in this way (i.e. *the* ~ D^0 ~ {f_1....f_n}).

Phonological strings mapping into primitive lexical items and strings mapping into complex decompositions exist in the same lexicon with no principled difference between the two. They are both stored in a dictionary structure with the phonological string as the key, and the feature set or morphological decomposition as value. It is possible to feed the algorithm with any string that exist in the lexicon. Thus, input strong *John * T * admire * Mary* generates in many cases the same outcome as *John admires Mary*, the only difference begin that steps (54)a–c are now omitted. Symbol T must, of course, exist in the lexicon as a key and map into a primitive lexical item containing tense information and other features that belong to this item. It is also possible to feed the model directly with morphological decompositions, which skips the step (54)a. In this way, the researcher can test the model with (possible or impossible, attested or unattested) words that do not exist in the lexicon. Thus, input *John admire#s Mary* will generate the same output as *John admires Mary*, assuming that the phonological word *admires* maps to *admire#s* in the same lexicon. In this way, it is possible to simulate morphological parsing. We can also test the model with words that are impossible or otherwise unattested, for example, what would happen if English adpositions were combined with Finnish-style possessive suffixes, or what happens if we combine English *that* with the third person singular agreement affix (e.g., *that#s*). Finally, we can bypass the whole content lexicon and feed the model with "general" lexical items standing for the major lexical categories, that is by strings such as D N T V D N (corresponding to *John admires Mary*) where D, N, T and V map to general determiner, noun, tense and verb units that have no specific content such as 'John' or 'admire'.

Consider the following figure which contains a screenshot of a small segment from the file defining the content lexicon, specifically, the some of the entries for the verb 'paint'.

```
206   maalatessa  :: maalaa-#v#ESSA/inf LANG:FI
207   maalattua   :: maalaa-#v#TUA/inf LANG:FI
208   maalaavan   :: maalaa-#v#VA/inf LANG:FI
209   maalata     :: maalaa-#v#A/inf LANG:FI
210   maalasi     :: maalaa-#v#T/fin#[V] LANG:FI
211   maalaa      :: maalaa-#v#T/fin LANG:FI
212   maalaa-     :: PF:maalaa- LF:paint V !COMP:ϕ SPEC:ϕ CLASS/TR LANG:FI
```

The first six items represent complex phonological words (keys in the lexicon dictionary) which map into morphological decompositions, strings of further entries in the same lexicon. These entries all begin with *maalaa-* which maps into a primitive lexical item, the verb 'to paint' (line 212). This is the root stem. Its features are listed on the line 212, among them its overt phonological shape (PF:maalaa-), idiosyncratic meaning (LF:paint), major lexical category (V) and then several selection features.[36] The decompositions are then followed by one or several morphemes all separate from each other by # , representing morpheme boundaries (symbol = represents clitic boundary). Symbol v denotes a grammatical head associated with transitivity and voice, symbols ESSA/inf and A/inf are infinitival heads; they all exist in the same lexicon as separate entries. Symbol [V] represents vowel lengthening and maps into the finite third person singular agreement morpheme in Finnish. These decompositions should match with the way the words are glossed or, stated in other way, they *are* the glosses provided in a format which the algorithm can interpret.

It is possible to have the same phonological string to map into two or more lexical entries. This creates *ambiguous words*, such as *bank*. When model confronts an ambiguous word, all its lexical entries are queued to the general recursive parsing algorithm and create their own parsing subtrees. This can be avoided, if it is not required for the research purposes at hand, by disambiguating the word manually in the input string. We could create, for example, two words *bank1* and *bank2* mapping into separate lexical entries and then use these symbols in the input. This makes the interpretation of the results clearer and easier when the irrelevant lexical entries are not contaminating the output.

The lexico-morphological component does not have access to the notion of complex constituent. It operates with linear objects of symbols matched and retrieved from the lexicon and which are then processed as linear objects. The lexical objects are fed into syntax as a linear stream, and

[36] Features are ultimately processed as "formal patterns" and are represented as strings.

only form complex constituents when syntax assembles them into complex heads and constituents.

4.9.2 Lexical items and lexical redundancy rules

Primitive lexical items that are stored in the lexicon (a dictionary data structure) are sets of features, which 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. 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.

Another source of lexical features comes from *lexical redundancy rules*. For example, the fact that transitive verbs can select object arguments need not be listed separately in connection with each transitive verb. This pattern is provided by redundancy rules which are stored in the form of feature implications '$F_1 \ldots F_n \rightarrow G_1 \ldots G_m$' stating that if a lexical item has features '$F_1 \ldots F_n$' it will also get features '$G_1 \ldots G_m$'. If there is a conflict between what is specified in the language-specific lexicon and in the redundancy rules, the language-specific lexicon wins. That is, the redundancy rules define what we can consider to be the default template or perhaps prototypical lexical items that are still subject to possible exceptions specified in the language-specific lexicon.

A repertoire of universal morphemes constitutes a third source. It contains elements like T, v and C. These are assumed to be present in all or almost all languages. Their properties too are modified by lexical redundancy rules.

One of the features in the language-specific lexicon linked with any item is its *language*. The feature is currently provided in the form [LANG:XX] where XX is a symbol for the language (e.g., FI = Finnish, EN = English, IT = Italian, HU = Hungarian and so on). As the name implies, this feature specifies the language (or dialect) to which the item belongs to. Because language is represented as a feature, it can enter into lexical redundancy rules. In this way, it is possible to

create *language-specific lexical redundancy rules* and, by using such rules, language-specific variations of common functional lexical items such as C, T or v.

Let us consider these assumptions by using an example, adpositions. Adpositions are represented by the general major lexical feature P, which is associated with a set of universal features by a lexical redundancy rule. For example, no adposition takes finite clause complements. Because these rules are specified as lexical redundancy rules, there can still be exceptions. Thus, if in some language there exist a special group of adpositions which take finite clause complements, it is possible override the lexical redundancy rule by providing the required selectional features in the language-specific lexicon.

Let us consider a more realistic example. While most adpositions require a noun phrase complement (*He ran towards me, *He ran towards*), there are exceptions such as *He lives near (me)* that can be specified in connection with the individual lexical items. In this case, then, there is a general pattern that is subject to a few individual exceptions. Finnish adpositions however differ from English adpositions in that the former, but not the latter, can exhibit agreement and occur in postpositional form (*lähellä minua* 'near I.par', *minun lähellä-ni* 'I.gen near-px/1sg'). This is something that can be modeled by relying on language-specific lexical redundancy rules of the form 'preposition + LANG:FI → agreement + postpositional configuration' which add the required features (whatever they are) into the Finnish adpositions. If such properties are only associated with a group of adpositions, then it is possible to form an additional feature representing the relevant grouping (much like we have intransitive and transitive verb stems) and add that feature to the redundancy rule.

4.9.3 Derivational and inflectional morphemes

Inflectional and derivational morphemes differ in how they are processed. Derivational morphemes are processed as described above: they are mapped into primitive lexical items, i.e. feature sets, and are streamed into syntax where they form complex heads and then head spreads via head reconstruction (§ 4.8.5). Inflectional features, too, are mapped into feature sets, but these

features are inserted *inside* an adjacent lexical item in word and, as a consequence, in the lexical stream. For example, the verb *admires* is decomposed into three elements *admire*#T#3sg by morphological decomposition in the lexicon, where the last element represents the third person agreement features. This is symbol mapped into the corresponding feature set, phi-features (singular, third person) in this case, which are inserted inside T as lexical features, thus *admire*#T#3sg ~ $T_{3sg}(V)^0$ ~ […[T_{3sg} […V…]]. If we specified in the lexicon that 3sg refers to a derivational element, then a separate agreement head such as Agr^0 would be generated. Sometimes there is evidence that agreement projects separate heads rather than just features, sometimes the issue is unclear (as in the case of Finnish possessive suffixes). Both hypotheses can be tested easily as just explained.

4.9.4 Lexical features

The catalog of lexical features undergoes changes as the model is being applied to new empirical phenomena. Some of the most important lexical features are explained in this section. Most of them can only be interpreted against the empirical data they were originally created to capture. All features are subject to the *feature conflict rule* which bans any derivation that contains a feature [F] and its counter feature [–F] at the LF-interface.

[PF:X] provides the phonological matrix of the element, currently just a string. [LF:X] provides its idiosyncratic semantic interpretation.

[COMP:L], [!COMP:L] and [–COMP:L] license, mandate or block complements of the type L, where L can be any feature inside the head of the complement but usually refers to one of the major lexical categories . The relevant notion of complement of α is the sister of α with the exclusion of [β^0 α] where β is primitive. The head is determined by labeling algorithm § 4.5. [( )COMP:*] applies the rule to any head, thus [!COMP:*] designates a functional head that has some complement.

[SPEC:L], [!SPEC:L] and [−SPEC:L] license, mandate or block specifiers of the type L, which prevents any non-adjunct constituent with a head containing feature L to occur at the edge of α . Notice that [−SPEC:L] is equivalent to [−L] due to the feature conflict rule. Feature [( )SPEC:*] applies the rule to any head, which creates a phenomenon resembling an unselective EPP requirement.

Features [(!)SPEC:φ] and [(!)COMP:φ] are of special interest and license or mandate referential arguments and are interpreted as assigning thematic roles.

[TAIL:L] requires that the element tails a head with feature L. [TAIL:L_1,...,L_n] requires a match with a set of features [TAIL:L_1,...,L_n].

[SELF:L], [!SELF:L] and [−SELF:L] license, mandate or block the head to have feature L . This feature become meaningful in cases where the derivation changes the feature content of some head α. We can use these features to block or require operations (or to "trigger them" in a reverse architecture). [−SELF:L] is equivalent to [−L].

[EF] licenses a nonthematic specific position. A nonthematic specifier at the edge of α is not assigned any thematic role and must therefore belong to a chain that has one member at a thematic position (excluding the expletives). This corresponds roughly to the notion of "standard EPP" as discussed in (Chomsky, 2008), although this feature by itself does not *require* that the position is filled, it only licenses it. In the present model, [EF] is interpreted as saying that the functional head belongs to a complex predicate and shares an argument with another head which is part of the same predicate. The reason it is nonthematic is because it is assumed that some other head assigns the thematic role, we do not want to create a situation where the separate parts of the complex predicate assign different thematic roles for the one and the same argument.

[Φ] signifies that Agree(α, X) § 4.8.8 occurred at α and that the head is linked with an independent local referential argument. [!SELF: Φ] then means that Agree(α, X) is mandatory, [−SELF: Φ] that it cannot occur. [−SELF: Φ] is equivalent to [−Φ] due to the feature conflict rule. [ΦPF] signifies

that overt agreement was detected in the input. Feature [−ΦPF] then blocks overt agreement because it leads into feature conflict. [ΦLF] signifies that phrasal Agree § 4.8.8 occurred, excluding pro-agreement. [−ΦLF] blocks phrasal Agree.

[p] signifies that the specifier of α was filled by phonologically overt material targeted for phrasal reconstruction.[37] [!SELF:p] will then correspond to the standard EPP feature, [−SELF: p] to an anti-EPP feature. [−SELF:p] is equivalent to [−p] due to the feature conflict rule.

Feature [null] is assigned to a head which is phonologically null.

[PHI:X:Y] denotes a phi-feature with type X and value Y.

4.10 Narrow semantics

4.10.1 Syntax, semantics and cognition

Semantics is the study 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." This set or structure contains things like persons, actions, thoughts or propositions as represented by the hearer and the speaker. This temporary semantic repository is called *global discourse inventory*. The discourse inventory can be accessed by global cognition, operations like thinking, decision making, planning, problem solving and others. Linguistic expressions, in turn, are utilized to introduce, remove and update entities in the discourse inventory. Consider the sentence *the horse raced past the barn*. We assume that the hearer projects the following entities into the global discourse inventory as s/he processes the sentence.

[37] It is assigned to any head α when an overt phrasal specifier marked for reconstruction is detected, see function *select_objects_from_edge* in the *phrase_structure.py.*

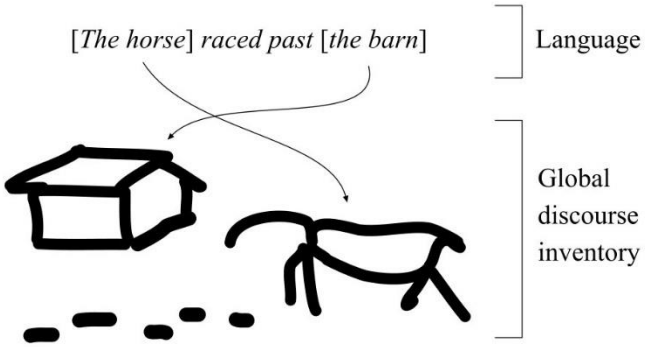


Figure 37. Narrow semantics projects semantic objects into the global discourse inventory.

We assume that this process of projecting and updating entities inside the global discourse inventory gives linguistic expressions their "meaning" in the narrow technical sense relevant to the present work.

The hypothesis that linguistic expressions provide a vehicle for updating the contents of the discourse inventory requires that there exists some mechanism which translated linguistic signals into changes inside the global discourse inventory. This mechanism, or rather collection of mechanisms, is called *narrow semantics*. It can be viewed as a gateway that encapsulates the syntactic pathway and mediates communication in and out. Cognitive systems that are outside of narrow semantics belong to global cognition and incorporate language-external cognitive processes. Narrow semantics is implemented by special-purpose functions or modules which interpret linguistic features arriving through the syntax-semantics interface. We can perhaps imagine language together with the narrow semantics as a cognitive system that grammaticalizes extralinguistic cognitive resources.

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 (Brattico, 2021a). Narrow semantics can therefore be also viewed as a gateway or "router," where the processing of different features is distributed

to different language-external systems or where the language faculty or syntactic processing pathway makes contact with other cognitive systems (see Figure 38).

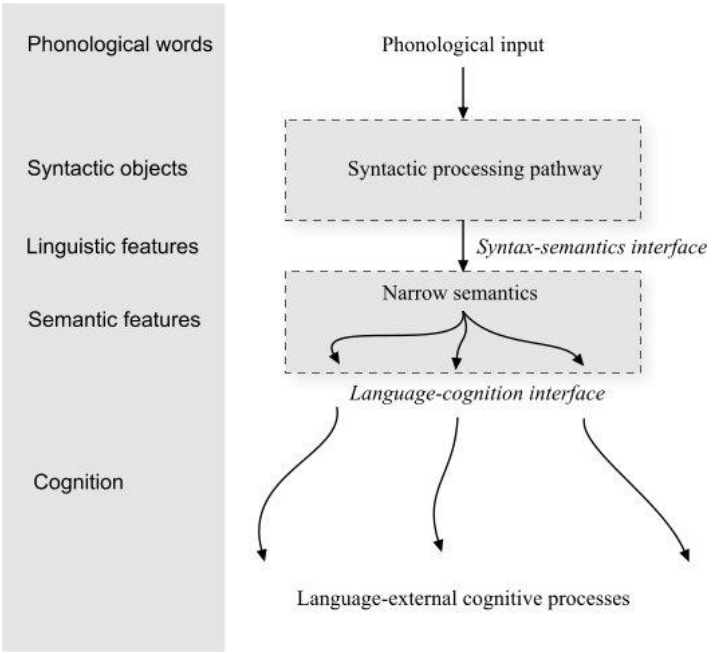


Figure 38. The general cognitive architecture. Narrow semantics bleeds input from language and feeds it in a transformed shape to other cognitive systems with which it makes contact. What these connections are or can depend on the innate neuronal architecture of the human brain.

A proper name such as *John* can be thought of as referring to a simple "thing" like an individual person. A pronoun like *he*, on the other hand, can refer in principle to several objects in the discourse inventory (*John$_1$ admires Simon$_2$, and he$_{1,2}$ is very clever*), and the same is true of quantifiers like *every man* or *two men*. Furthermore, expressions like *no one* do not refer to anything, yet they, too, have meaning. Even *John* can be ambiguous if there are several men with the same name. The general problem is that there is nothing in the expression itself that determines unambiguously what it denotes, so the listener must perform some type of selection and/or guessing. To handle this, all expressions are first linked to unambiguous semantic entries inside narrow semantics, representing their intrinsic semantic properties, and these intermediate representations are transformed into actual denotations that point into semantic objects in the

global discourse inventory accessed by other cognitive processes. To illustrate, consider a short conversation *The horse raced past the barn; it was very fast*. The first sentence will establish that there are two things in the global discourse inventory the sentence speaks about: the horse and the barn. The next sentence then makes a claim about some "it" that we must link with something. This pronoun can denote four things in this conversation: the horse, the barn, the whole event, or a third entity not yet mentioned, as shown in Figure 39.

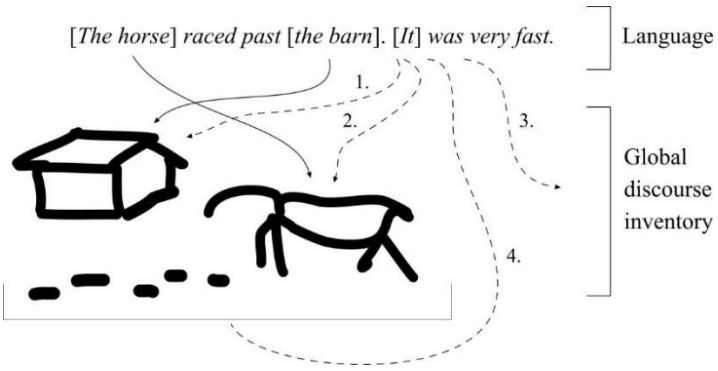


Figure 39. Possible denotations for expression *it*.

Narrow semantics calculates possible denotations by using the properties of the referring expression itself (e.g., nonhuman, singular, third party in the conversation) and what is contained in the global discourse inventory at the time when the expression is interpreted (the horse, the barn, the event, a possible third entity). The truth-value of the sentence, and thus its ultimate meaning in a context, is calculated when all referential expressions are provided some denotation. This is called *assignment*. The most plausible assignment in this case is one in which *the horse* denotes the horse, *the barn* denotes the barn, and *it* denotes the horse as well. The assignment in which *it* denotes the barn is implausible, but possible in principle. The model provides all possible assignments and their rankings as output. Assignments are calculated at the language-cognition interface.

When a linguistic feature such as [SG] 'singular' is transformed into a formal understood by global cognition, what we mean is that the formal signal representing that feature in the linguistic output will activate a corresponding signal or representation (representing, say, 'one') inside the

extralinguistic cognitive system. In the case of quantifiers such as *some*, the mapping is more complex but the principle is the same. This quantifier signals that we are supposed to select some (but no matter what and how many) objects from the global discourse inventory. I assume that the operation of 'selecting some' is part of the human cognitive repertoire accessed by narrow semantics, and that this is the reason a quantifier (or a lexical feature corresponding to it) can exist.

4.10.2 Argument structure

Argument structure refers to the way referential arguments are organized syntactically and semantically around their predicates. Let us consider some of the fundamental properties of this system. Consider the situation depicted in Figure 8.

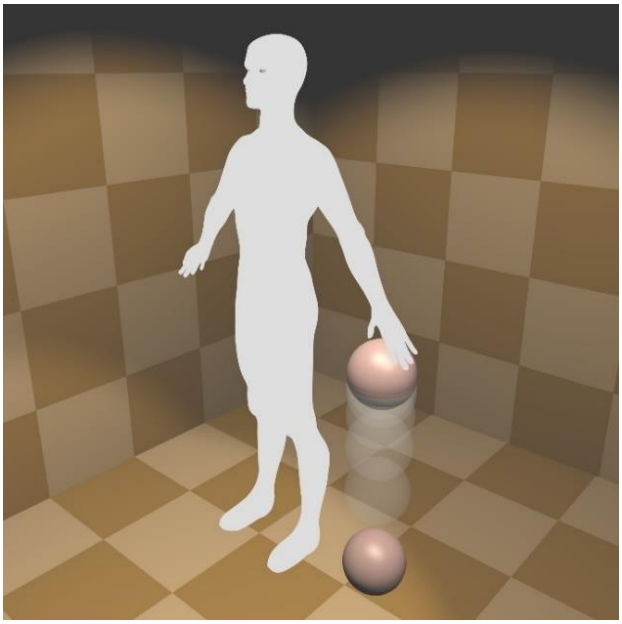


Figure 8. A non-discursive, perceptual or imagined representation of the meaning of the sentence *John dropped the ball*.

Imagine you have an non-discursive (analogous, continuous) experience of a person dropping a ball. We assume that instead of a static figure, you are looking at a dynamic scene that evolves over time. The visual-conceptual experience will be structured in such a way that we perceive the situation as involving a spatiotemporally continuous person, the ball and an event where the

person drops the ball and the ball falls as a consequence. Finally, the event ends when the ball touches the floor. These are the entities that would exist, or come into existence, in the discourse inventory at the moment we experience the event. The grammatical elements that make up the sentence *John dropped the ball* grammaticalize these conceptual objects, in this case John, the ball and the event of dropping expressed by the verb. Part of the meaning emerges from the structure into which they have been embedded. Lexical items and their structure are linked with the contents of the discourse inventory and, ultimately, the everyday human experience. Let us consider the grammatical representation of the sentence (55) and how the different parts may correlate with the conceptual experience depicted in Figure 8.

(55) [$_{TP}$ John$_1$ [$_{TP}$ T$_{pst}$ [$_{vP}$ __$_1$ [$_{vP}$ v(drop) [$_{VP}$ fall [the ball]]]]]]

The combination of the object the ball and the verb, the lower VP, correspond to a subevent where the ball falls. Thus, a bare verb together with an object is typically interpreted as denoting an event that involve one participant. This is combined with the small verb v and the agent argument John, which is interpreted as the causer of the subevent. That is, John is the agent of the event because he causes the ball to fall. The combination of *fall* + v will be the transitive verb *drop* 'cause to fall'. Consequently, the argument at SpecvP is interpreted as the agent of the event, while the argument inside the VP is interpreted as the patient that must undergo the event caused by the agent. These interpretations are created when the representation arriving at the LF-interface is interpreted semantically by narrow semantics. Tense T adds tense information to the event, but the argument at SpecTP will not get a secondary thematic interpretation. In sum, the lexical items and their surrounding structures map into entities in the discourse inventory and human experience, in this case objects, events, causation and temporal properties. The elements must be organized in some specific way in order for this interpretation to succeed, and the formal selection restriction features present in the lexicon guide the syntactic processing pathway towards

solutions which satisfy these ultimately semantic requirements grounded in the structure of human experience.[38]

4.10.3 Predicates, arguments and antecedents

Predicates are characterized by the fact that they must be linked with arguments (§ 4.8.8). A predicate is defined as such by unvalued phi-features φ_, representing an "unsaturated argument" in the Fregean sense. The same property can be represented by feature [ARG] suggesting "needs an argument" which under the current model automatically generates an uninterpretable phi-set φ_ to the lexical item.

A predicate α is linked to an argument at the LF-interface/narrow semantics by finding first and closest argument from its *edge*, where edge refers to α's (i) own feature content, (ii) complement, (iii) specifier(s) and finally to (iv) any constituent within the upward path (22) § 4.6. The search space defined by (i-iv) is explored in this order, and the first potential argument found will be selected. That is, if the head contains a sufficiently rich pro-element (e.g., in virtue of strong agreement), that will represent the argument (56)a; if not, then if the head has a referential complement, it is selected (56)b; if not, then the specifier is examined (56)c; if no suitable referential specifier exists, then the whole upward path is explored (56)d–e.

(56) a. Lähde-n kotiin.
leave-prs-1sg home
'I leave home'

b. Kohti talo-a.

[38] Categories like 'causing the ball to fall' or even 'the ball' are not dichotomous categories one could easily 'program' in Python. What happens, instead, is that the speakers and hearers project these categories to the world and/or to their own experiences. John is conceptualized as the cause of the event because we perceive, correctly or incorrectly, that he initiates the action. Yet merely willing the ball to fall is not sufficient; John must also perform actions that allow the gravity to take over. If an external agency controls John's mind, then the sentence *John dropped the ball* would be false if *John* refers to the person, but true if we use *John* to refer to his physical body, as in the sentence *the tree dropped its leaves*. In this sense sentences provide "perspectives" to reality by conceptualizing parts of it.

towards house-par

'towards the house'

c. Sitä päätti [Pekka (taas) [lähteä aikaisin]].

expl decide-pst-3sg Pekka again leave early

'Pekka decided to leave early again.'

d. Pekka sanoi että __ halua-a lähteä.

Pekka said that want-prs.3sg leave

'Pekka said that he wants to leave.'

e. Pekka wants to sleep.

Intuitively we can think that the argument of the predicate must be "around" it at the LF-interface representation. Failure to find an argument from αP results in a situation where the argument is searched from a nonlocal domain (56)d–e, capturing control (Brattico, 2021b).

The argument-predicate mechanism interacts with Agree-1. When the argument is located *inside* the complement, it is invisible for the predicate-argument mapping rule. Agree-1 detects it and copies its phi-features to the head, which is some cases is sufficient to interpret the argument (rule i). In some cases this is not sufficient, or it does not occur at all, and the argument must be literally copied to the specifier position of α in order to be visible for α (rule iii). This captures the EPP phenomenon in the current model and creates A-chains § 4.8.4. The idea of capturing the EPP/Agree complex by relying on a theory of predication comes from by (Rothstein, 1983) and has been developed by many others, for example (Chomsky, 1986, p. 116; Heycock, 1991; Kiss, 2002; Rosengren, 2002; Williams, 1980). The notion of edge that plays a key role in the present model is developed on the basis of (Chomsky, 2008).

4.10.4 The pragmatic pathway

In addition to the syntactic processing pathway, there exists a separate pragmatic pathway that monitors the incoming linguistic information and uses it to create pragmatic interpretations for the illocutionary act and/or communicative situations associated with the input sentence. The

same pragmatic pathway computes topic and focus properties to the extent that they have not been grammaticalized and/or are based on general psychological characteristics of the communicative situation. This means that what linguistics describe as 'information structure' is partitioned into an interpretative extralinguistic pragmatic component and a syntactic (grammaticalized) component.

The pragmatic pathway works by allocating attentional resources to the incoming expressions and the corresponding semantic objects and by notifying if an element is attended in an unexpected ('too early' versus 'too late') position. The current implementation relies on two syntax-pragmatics interfaces to handle these cases. The first interface occurs very early in the processing pipeline – at the lexical stream currently – and registers all incoming referential expressions and allocates attentional resources to them. Another interface is connected to the component implementing discourse-configurational word order variations during transfer. It responds to situations in which some expression occurs in a noncanonical 'too early' or 'too late' position, and then generates the corresponding pragmatic interpretations 'more topical' and 'more focus-like', respectively. The results are shown in the field "information structure" in the output. By positioning the expression into a certain noncanonical position the speaker wanted specifically to control the attentional resource allocation of the hearer, and the hearer then infers that this must be the case.

Properties of the pragmatic pathway can also grammaticalize into *discourse features* that are interpreted by the pragmatic pathway. Discourse features are marked by [DIS:F]. Thus, it is possible for language also to mark topics or focus elements by using special features (which may also be prosodic). The idea that notions generated by language-external cognitive systems grammaticalize inside the syntactic pathway is one of the core principles of the semantic system.

4.10.5 Operators

Operators are elements that create operator-variable dependencies, usually (perhaps exclusively) represented by Ā-chains § 4.8.2, 4.8.3. Interrogative clauses (57) represent a prototypical example of operators and operator-variable structures.

(57) What did John found __?

Operators are processed inside an operator-variable module by linking lexical elements containing operator features with a scope-marker that determines the propositional scope for that particular operator. The operation is triggered when narrow semantics sends a lexical element containing an operator feature such as [*wh*] for the operator-variable module for interpretation. The scope marker is closest lexical head inside the upward path with [*wh*][FIN]. The finite operator cluster [*wh*][FIN] is created during reconstruction. The relevant configuration is illustrated by (58).

(58) What did$_{wh,\text{fin}}$ John found (what)?

Here the operator feature at the in situ interrogative pronoun *what* triggers the binding mechanism, which locates the scope marking element at C having features [*wh*] and [FIN] and generates the meaning ‘what x: John found x’.

4.10.6 *Binding*

All referential expressions such as pronouns, anaphors or proper names must be linked with some object or objects in the discourse inventory so that both the hearer and speaker know what is “talked about.” This is a nontrivial problem for language comprehension, because all such expressions are ambiguous. Even a proper name such as *John* can refer to any male person in the current conversation whose name is or is assumed to be John. The model restricts possible denotations by using whatever lexical features are available in the lexical items themselves and whatever is available in the global discourse inventory. For example, a proper name *John* can only denote a single male person what that name. In the same way, *she* cannot denote a male person. Both can denote something that does not (yet) exist in the global discourse inventory.

Some referential expressions also impose structure-dependent restrictions on what they can denote. Reflexives like *himself* must be coreferential with a nonlocal antecedent (59), *him* cannot denote a too local antecedent (60), and R-expressions such as proper names must remain free (61).

(59) a. John$_1$ admires himself$_{1,*2}$.

b. *John's$_1$ sister admires himself$_1$

(60) a. John$_1$ admires him$_{*1,2}$

b. John's$_1$ sister admires him$_{1,2}$

(61) a. He$_1$ admires John$_{*1,2}$.

b. His$_{1,2}$ sister hates John$_1$.

Binding theory is concerned with the conditions that regulate these properties. Since assignments are computed at the language-cognition interface, whatever mechanism drives binding must regulate what takes place there. This is implemented technically by assuming that nominal expressions (e.g., anaphora, pronouns, R-expressions) contain features that provide "instructions" for a cognitive system that then filters possible assignments, as shown in the data above.

5 Performance

5.1 Human and engineering parsers

The linear phase theory is a model of human language comprehension. Its behavior and internal operation should not be inconsistent with what is known independently concerning human behavior from psycholinguistic and neurolinguistic studies and, when it is, such inconsistencies must be regarded as defects that should not be ignored, or judged 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.

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. The image is modified from (Brattico, 2023b).

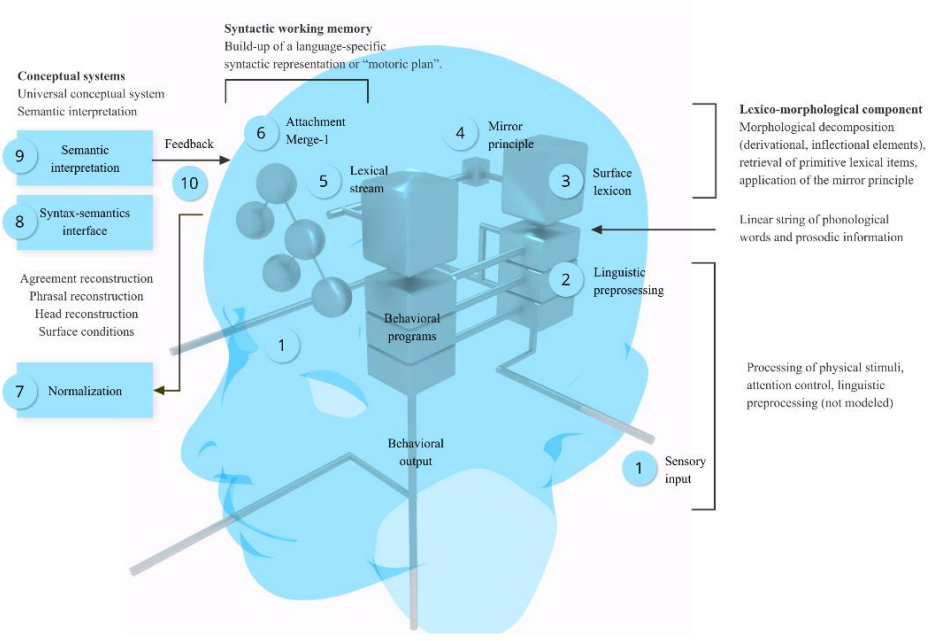


Figure 32. Components of the model and their approximate locations in the human brain. See the main text for explanation.

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 a tokenized string (2) as 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)(§ 4.3). Preprocessing is done in lower-level sensory systems that rely on the various modules within the brain stem. Activated lexical items are streamed into syntax (5). 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)(§ 4.1). It is possible that these same regions implement transfer (§ 4.8), 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. The architecture assumes that the endpoint of syntactic processing makes contact with other cognitive systems, possibly anywhere in the brain, via narrow semantics.

5.3 Cognitive parsing principles

5.3.1 Incrementality

The linear phase algorithm is incremental: 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 later, and no word is examined before other words that come 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 being developed at any given point 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 a wrong decision on the basis of incomplete local information, which shows that the parser does not wait for the appearance of further words before making its decisions. This can be seen from (62), 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.

(62)

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 in a dead end when processing the last word *fell*, backtracks, and consumes additional cognitive resources before it finds the correct analysis. The following shows how the algorithm derives (b). Step (11) contains the first dead end and starts the backtracking phase.

```
1   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 before the simulation trial. 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 garden paths.

I do not assume that the backtracking operation visible in the example above is entirely realistic from the psycholinguistic point of view. There are two reasons why it exists. One is that by performing systematic backtracking we let the model to explore the complete parsing tree. If we only generated the first solution, spurious secondary solutions might escape our attention. It is not untypical that the model finds ungrammatical and/or wrong secondary solutions, revealing that it is using a wrong competence model. The second reason is that backtracking gives us important information concerning the model's performance. We can compare the amount of computational resources spend in processing different types of constructions and/or different algorithmic solutions. In addition, realistic language comprehension by the human brain in connection with canonical sentences is seldom subject to any garden pathing, so we can at least aim for a model that finds the correct solution immediately, at least in those cases.[39]

[39] It is possible, in fact a worthwhile goal, to add a realistic backtracking model on the top of the systematic backtracking as an optional component. I suspect that real speakers solve garden path problems by starting the parsing from the beginning with additional noise added to the decision mechanism.

There is one situation in which incrementality is violated: inflectional features are stored in a temporary working memory buffer and enter the syntactic component inside lexical items. The third person agreement marker -s in English, for example, will be put into a temporary memory hold, inserted inside the next lexical item, which then enters syntax. The element therefore stays in the memory a very 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 where the syntactic working memory holds two or more phrase structures that are not connected to each other by means of some grammatical dependency. Adjunct structures are attached to their host structures loosely: they are geometrical constituents inside their host constructions, observing connectness, but invisible to many computational processes applied to the host (§ 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. The current parsing model is serial in the sense that the predicted cognitive cost is computed in this way by adding the cognitive costs from each individual operation together. 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. 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, most computational operations implemented by the linear phase model must be executed in an exact 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 which requires 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 (63) is first attached to the lowest possible solution, and only if that solution fails, to the nonlocal node.

(63) 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 (§ 6.3.4). If no choice is provided, bottom-up algorithm is used by 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 (§ 4.3, 6.3.2). 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 (§ 6.3.4). 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 (§ 6.3.4).

5.3.7 Working memory

There is psycholinguistic evidence 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 must be correct. The user can activate the working memory or knock it off by changing the study parameters in a manner explained in § 6.3.4 and in this way see what its effects are.

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 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. All filtering and ranking principles cease to apply to it. Finally, it is assumed that re-activation of a dormant constituent accrues a cognitive cost, 500ms in this version. 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.

5.3.8 Conflict resolution and weighting

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

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 summing predictive cognitive costs from 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). This metric takes sentence length into account. Resource consumption is summarized in the resource output file (§ 6.4.6) that lists each sentence together with the number of all computational operations (e.g., Merge, Agree, Move) consumed from the reading of the first word to the outputting of the first legible solution. The file uses 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.

5.5 A note on implementation

Most of the performance properties are implemented in their own module *plausbility_metrics.py* which determines how the attachment solutions (§ 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 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 uses the ordered list to organize the 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.

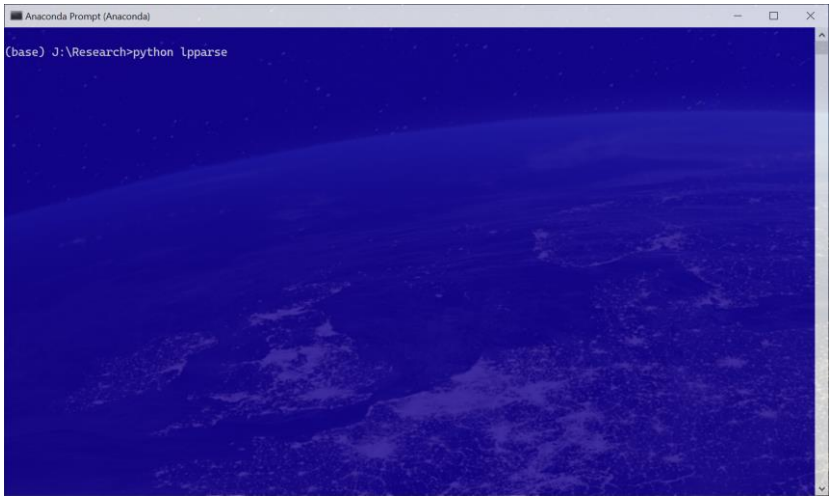
6 Inputs and outputs

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). 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 this. 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 as shown below:



This will parse all the sentences from the designated test corpus file in a study folder. 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 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, it will try to locate this file from the installation directory. Usually this file is present in any version currently being developed. If the file is not found, default values will be used. Any parameter that is defined in the *config_study.txt* can 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 as it exists in my local machine at the time of writing.

```
author= Anon
year= 2023
date= March
study_id= 15
study_folder=           language data working directory/study-16-EPP
lexicon_folder=         language data working directory/lexicons
test_corpus_folder=     language data working directory/study-16-EPP
test_corpus_file=       EPP_corpus.txt

# GENERAL SIMULATION PARAMETERS
only_first_solution = True
ignore_ungrammatical_sentences = False
console_output = Full
stop_at_unknown_lexical_item = False
logging = True
check_output = True

# DATA COLLECTION
datatake_full = False

# IMAGE PARAMETERS
datatake_images = False
image_parameter_show_words = True
image_parameter_nolabels = False
image_parameter_spellout = False
image_parameter_show_sentences = False
image_parameter_show_glosses = True
image_parameter_mark_adjuncts = False
image_parameter_stop_after_image = False
image_parameter_show_ϕPs_as_DPs = True
image_parameter_show_ϕPs_as_NPs = False
image_parameter_ignore_internal_structure_of_DPs = False
#EPP study
show_features = ΦPF,-SELF:Φ,!SELF:Φ,!SELF:p,EF,!COMP:ϕ,!SPEC:ϕ

#Finnish syntax study
#show_features = OP:Q

# UG COMPONENTS
#   Agree
#   standard = standard theory (Chomsky 2000, 2001, 2008)
#   revised = current best formulation
#   variation1 = revised formulation without PIC, only select closest
#   variation2 = revised formulation but with added specifier (edge) agreement as a last resort
#   variation3 = revised formulation with only specifier-head agreement
#   variation4 = revised formulation with each phi-features looking for its own goal
Agree = revised

#   Semantics
calculate_assignments = True
calculate_pragmatics = True
calculate_thematic_roles = True
project_objects = True
generate_argument_links = True
calculate_predicates = True

# PARSING HEURISTICS
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 semicolon. 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. If a key or parameter is missing, the 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, an error occur. 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 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 runs several studies.

Recall that the key cannot contain white spaces, but 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. When the user launches the program, module *__main__.py* is executed. This module takes care of reading and interpreting the command line parameters, and it can be used to diverge the execution to different modules based on command line parameters. The first component that belongs to the model itself is the main script *main.py* responsible for running one study. 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 test corpus with the parser, and processes and stores the results. The architecture is illustrated in Figure 12. The code for the main script is explained in § 6.5.

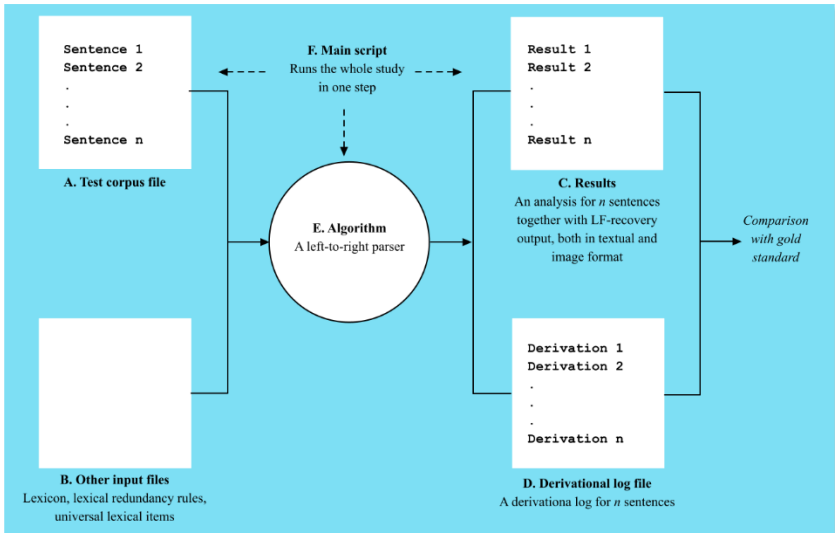


Figure 12. Relationships between the input, main script, linear phase parser and the output. Only two output files are shown, the main results file and the derivational log file.

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 stored as raw data associated with each study. The user cannot interfere the execution.

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. It contains the empirical theory. The program also 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.

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 a 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 Structure of the input files

6.3.1 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 white space 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. 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 a fully disambiguated lexicon could be considered an option.

Special symbols are used to render to output more readable and to help testing. Symbols # and single quotation (‘) in the beginning of the line are read as introducing comments and are ignored. This allows the user to write glosses below the test sentences. 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. Both are illustrated in the screenshot from the corpus file below.

```
56    & 1.2 Finnish high complementizer
57
58        & Grammatical
59
60            Pekka sanoi että Merja tekee työtä
61            'Pekka.nom say.prs.3sg that Mereja.nom make.prs.3sg work.par'
62
63        & Ungrammatical (illicit subject/object at SpecCP)
64
65            Pekka sanoi Merja että tekee työtä
66            'Pekka.nom say.prs.3sg Merja.nom that make.prs.3sg work.par'
67
68            Pekka sanoi työtä että Merja tekee
69            'Pekka.nom say.prs.3sg work.par that Merja.nom make.prs.3sg'
```

The above figure contains labels *& Grammatical* and *& Ungrammatical* which specify the grammaticality status of the sentences that follow the comment up to the next grammaticality specification. When the simulation is executed, these symbols are used to produce an error log file which contains a list of all sentences from the corpus file that were judged wrongly by the algorithm. The algorithm produces this file by matching the grammaticality labels with its own

judgments. This allows the researcher to evaluate the correctness of the algorithm without performing detailed file comparisons.

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 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 useful if the user wants to examine the processing of only one sentence (input sentences can also be provided from the command prompt).

```
56    & 1.2 Finnish high complementizer
57
58        & Grammatical
59
60            Pekka sanoi että Merja tekee työtä
61            'Pekka.nom say.prs.3sg that Mereja.nom make.prs.3sg work.par'
62
63        & Ungrammatical (illicit subject/object at SpecCP)
64
65        %   Pekka sanoi Merja että tekee työtä
66            'Pekka.nom say.prs.3sg Merja.nom that make.prs.3sg work.par'
67
68            Pekka sanoi työtä että Merja tekee
69            'Pekka.nom say.prs.3sg work.par that Merja.nom make.prs.3sg'
```

If the user wants 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= will be processed).[40]

It is possible to feed the input sentences to the model as individual sentences or as part of a larger conversation. A conversation is defined as a sequence of sentences which share the same global discourse inventory. To create a conversation between two sentences, use semicolon at the end of

[40] It is possible to use several =START= and =STOP= commands. 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.

the first sentence. The result of this is that the semantic objects instructed by the first sentence will be available as denotations for the expressions in the second (64).

(64) a. John$_1$ met Mary$_2$;

b. He$_{1,3}$ admires her$_{2,4}$.

Any number of sentences can be sequences into a conversation. If the sentence does not end with a semicolon, it is assumed that the conversation ended and the global discourse inventory is reset when the next sentence is processed. Thus, if sentence (64)a did not end with the semicolon, pronouns *he* and *her* in sentence (b) can no longer refer to them. Conversations can be used to create discourse contexts for test sentences.

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

The main script uses three lexical resource files that are by default called *lexicon.txt*, *redundancy_rules.txt* and *ug_morphemes.txt*. The first contains language specific lexical items as explained in § 4.9, the second a list of universal redundancy rules and the last a list of universal morphemes. Figure 15 illustrates the language-specific lexicon.

```
177 lähtemässä* :: lähte-#MA.ine/inf*#P* LANG:FI
178 lähtemästä  :: lähte-#MA.ela/inf LANG:FI
179 lähtemättä  :: lähte-#MA.abe/inf LANG:FI
180 lähtemällä  :: lähte-#MA.ade/inf LANG:FI
181 lähteäkse   :: lähte-#KSE/inf LANG:FI
182 lähteäkseen :: lähte-#KSE/inf#An LANG:FI
183 lähteäkseen* :: lähte-#A/inf*#KSE/inf*#An LANG:FI
184 lähtien     :: lähte-#E/inf LANG:FI
185 lähtiessä   :: lähte-#ESSA/inf LANG:FI
186 lähtiessään :: lähte-#ESSA/inf#An LANG:FI
187 lähdettyään :: lähte-#TUA/inf#An LANG:FI
188 lähdettyä   :: lähte-#TUA/inf LANG:FI
189 lähde       :: lähte-#T/fin LANG:FI
190 lähde       :: lähte-#T#default LANG:FI
191 lähti       :: lähte-#T/fin#[V] LANG:FI
192 lähdettiin  :: lähte-#impass#T/fin#tVVn LANG:FI
193 lähte-      :: PF:lähte- LF:leave CLASS/INTR V SPEC:ϕ -COMP:ϕ COMP:P LANG:FI
194 lle         :: PF:P(-lle) LF:for P -E +SEM:directional -ΦPF TAIL:V LANG:FI
195 lla         :: PF:P(-lla) LF:by P -ΦPF TAIL:T TAIL:*V LANG:FI
196 lukemalla   :: luke-#v#MA.ade/inf LANG:FI
197 lukee       :: luke-#v#T/fin#[V] LANG:FI
```

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 :: 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 morphological decomposition. If the element designates a primitive (terminal) lexical item, 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. 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 lexical streaming. The following screenshot shows the definitions for Finnish finite agreement affixes.

```
# Finite agreement
n       ::  inflectional ΦPF Φ EF -Inf PHI:NUM:SG PHI:PER:1 PER PHI:DET:DEF PHI:HUM:HUM LANG:FI
t       ::  inflectional ΦPF Φ EF -Inf PHI:NUM:SG PHI:PER:2 PER PHI:DET:DEF PHI:HUM:HUM LANG:FI
[V]     ::  inflectional ΦPF Φ EF -Inf PHI:NUM:SG PHI:PER:3 PER LANG:FI
mme     ::  inflectional ΦPF Φ EF -Inf PHI:NUM:PL PHI:PER:1 PER PHI:DET:DEF PHI:HUM:HUM LANG:FI
tte     ::  inflectional ΦPF Φ EF -Inf PHI:NUM:PL PHI:PER:2 PER PHI:DET:DEF PHI:HUM:HUM LANG:FI
vAt     ::  inflectional ΦPF Φ EF -Inf PHI:NUM:PL PHI:PER:3 PER LANG:FI
tVVn    ::  inflectional ΦPF Φ EF -Inf PHI:NUM:PL PHI:PER:1 PHI:DET:DEF PER LANG:FI
```

A lexical feature is a string, essentially a formal 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.[41] The file *ug_morphemes.txt* is structured in the same way but contains universal morphemes such as T and v. The following screenshot shows the definitions for Finnish infinitival heads which we assume are derivatives of a universal set (currently not modelled).

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

```
# Finnish infinitivals
A/inf       :: A/inf       EF  Inf     ?ARG -ΦPF              PF:-(t)A    LF:to -SPEC:N -SPEC:Neg/fin SEM/desired_event
VA/inf      :: VA/inf      EF  Inf T   ARG !SELF:Φ -SELF:Φ*    PF:-vAn     LF:to -SPEC:N !COMP:*  PROBE:V LANG:FI
ESSA/inf    :: ESSA/inf    EF  Inf Adv ARG                     PF:-essA    LF:while !COMP:*  PROBE:V LANG:FI TAIL:T
TUA/inf     :: TUA/inf     EF  Inf Adv ARG !SELF:Φ -SELF:Φ*    PF:-tUA     LF:after !COMP:*  PROBE:V LANG:FI TAIL:T
KSE/inf     :: KSE/inf     EF  Inf Adv ARG -SELF:ΦLF           PF:-kse     LF:for !COMP:*  PROBE:V LANG:FI TAIL:T
E/inf       :: E/inf       EF  Inf Adv ARG                     PF:-en      LF:by !COMP:*  PROBE:V LANG:FI TAIL:T
MA.ine/inf  :: MA.ine/inf  EF  Inf Adv ARG                     PF:-mAssA   LF:in !COMP:*  PROBE:V LANG:FI TAIL:V
MA.abe/inf  :: MA.abe/inf  EF  Inf Adv ARG                     PF:-mAttA   LF:without !COMP:*  PROBE:V LANG:FI TAIL:T
MA.ade/inf  :: MA.ade/inf  EF  Inf Adv ARG                     PF:-mAllA   LF:by !COMP:*  PROBE:V LANG:FI TAIL:V
MA.ela/inf  :: MA.ela/inf  EF  Inf Adv ARG                     PF:-mAstA   LF:without !COMP:*  PROBE:V LANG:FI TAIL:CLAS
MA.ill/inf  :: MA.ill/inf  EF  Inf     ARG                     PF:-mAAn    LF:to SEM/desired_event !COMP:*  PROBE:V LANG
```

6.3.3 Lexical redundancy rules

Lexical redundancy rules are provided in the file *redundancy_rules.txt* and 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. Below is a screenshot from the file containing the lexical redundancy rules.

```
FORCE   :: -ARG -EF Fin !COMP:* !PROBE:Fin COMP:T/fin CO
C/fin   :: -ARG Fin C -COMP:C/fin -COMP:T/prt COMP:T/fin
C       :: -SPEC:MA/A EF
T/fin   :: ARG EF !SELF:Φ T Fin !COMP:* -SPEC:C SPEC:T/p
T/prt   :: ARG Fin T EF !COMP:* COMP:v COMP:V COMP:T/prt
T       :: ARG COMP:* COMP:v COMP:V -COMP:ϕ -COMP:D -COM
Neg/fin :: ARG EF Fin NEG COMP:T/prt COMP:T -SPEC:T -SPE
v       :: ARG -ΦPF ASP !COMP:* COMP:V -SPEC:v -SPEC:T/f
D       :: ϕ -ARG !COMP:* -COMP:D -COMP:v -COMP:V -COMP:
D/rel   :: ϕ D -ARG OP OP:REL COMP:* -COMP:C/fin -COMP:ϕ
N       :: -COMP:D -COMP:EXPL -COMP:VA/inf -COMP:T/fin -
V       :: ARG -ΦPF ASP -SPEC:T/fin -SPEC:FORCE SPEC:Adv
P       :: ARG COMP:ϕ -COMP:V -COMP:MA/A -SPEC:Neg -COMP
Adv     :: -SPEC:N -SPEC:FORCE -SPEC:Neg/fin -SPEC:T/fin
A       :: COMP:0 TAIL:ϕ -SPEC:A
Inf     :: TAM -COMP:T/fin -COMP:FORCE -COMP:C/fin -SPEC
n       :: ϕ EF ARG Inf !COMP:* COMP:0 COMP:V COMP:v
0       :: SPEC:ϕ 2SPEC -COMP:ϕ
Num     :: ϕ COMP:ϕ COMP:N COMP:n -COMP:D -COMP:QN
DET     :: ϕ COMP:ϕ -COMP:DET
QN      :: ϕ COMP:ϕ COMP:N COMP:Num
ϕ       :: COMP:N -SPEC:T -SPEC:P COMP:D/rel !COMP:*
```

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 the "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]).

6.3.4 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. See § 6.2.

6.4 Structure of the output files

6.4.1 Results

The name and location of the results file is determined when configuring the main script in *config_study.txt*. The default name is made up by combining the test corpus name together with *_results.txt.* Each time the main script is run, the default results file is overridden. 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 below shows one grammatical analysis (line 11) together with semantic interpretation (lines 13–20), contents of the global discourse inventory (lines 23–24, in simplified format) and performance metrics (lines 27–29).

```
1.  Seine flows towards Paris

    [[D Seine]:1 [T [__:1 [flow [towards [D Paris]]]]]]

    Semantics:
    Thematic roles: ['Agent of V°(flow): [D Seine]', 'Patient of P°(towards): [D Paris]']
    Predicates: ['T°: [D Seine]', 'flow°: [D Seine]', 'towards°: pro/[D Paris]']
    Speaker attitude: ['Declarative']
    Assignments:
    [D Seine] ~ 2, [D Paris] ~ 6, Weight 1
    [D Seine] ~ 6, [D Paris] ~ 2, Weight 1
    Information structure: {'Marked topics': [], 'Neutral gradient': ['[D Seine]', '[towards [D Paris]]', '[D Paris]'],

    Discourse inventory:
    Object 2 in GLOBAL: [D Seine]
    Object 6 in GLOBAL: [D Paris]

    Resources:
    Total Time:1027, Garden Paths:0, Merge:6, Head Chain:7, Phrasal Chain:4, Feature Chain:0,
    Agree:0, Feature:0, Scrambling Chain:0, Extraposition:2, Last Resort Extraposition:0,
    Mean time per word:256, Merge-1:18,
```

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. If the test corpus contains grammaticality labels *& Grammatical* and *& Ungrammatical*, the same information is also generated to the *error reports* file.

6.4.2 The log file

The derivational log file, created by default by adding *_log* into the name of the test corpus file, contains a more detailed report of the computational steps consumed in processing each sentence in the test corpus and of the semantic interpretation. 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, transfer operation applied to the final structure when no more input words are analyzed and many aspects of semantic interpretation. Intermediate left branch transfer operations are not reported in detail. The beginning of a log file is illustrated in the figure below.

```
3  #1. Seine flows towards Paris
4  ['Seine', 'flows', 'towards', 'Paris']
5
6       1. ['Seine', 'flows', 'towards', 'Paris']
7
8      Morphological decomposition of /Seine/ ~ ['Seine-', 'D$', 'sg$', '3p$', 'def$']
9      Next affix [def]
10     Next affix [3p]
11     Next affix [sg]
12     Next morph [D] ~ D°
13
14     Next morph [Seine-] ~ N°
15
16         Ranking:
17         (1) [D↓+ N°](⇒ #0x1#)
18         #0x1#
19         Merge(Seine) => D(N)
20
21     Morphological decomposition of /flows/ ~ ['flow-', 'T/fin$', '[-s]$']
22     Next affix [[-s]]
23     Next morph [T/fin] ~ T°
24
25         Filtering and ranking merge sites...+Spec selection for D(N)...(100) {'T/fin'}-Comp selection for D(N)...(-100)
26         Ranking:
27         (1) [D(N)↓+ T°](⇒ #0x2#)
28         (2) [D(N) + T°](=> #0x3#)
29         #0x2#
30         Merge(T) => [[D Seine] T]
31
32     Next morph [flow-] ~ V°
33
34         Ranking:
35         (1) [T↓+ V°](⇒ #0x4#)
36         #0x4#
37         Merge(flow) => [[D Seine] T(V)]
38
39     Next morph [towards] ~ P°
40
41         Filtering and ranking merge sites...-Spec selection for [[D Seine] T(V)]...(-100) +Comp selection for T(V)...(100)
42         Ranking:
43         (1) [T(V) + P°](=> #0x5#)
44         (2) [T(V)↓+ P°](⇒ #0x6#)
45         (3) [[[D Seine] T(V)]↓+ P°](⇒ #0x7#)
46         #0x5#
47         Merge(towards) => [[D Seine] [T(V) towards]]
```

This segments contains the Merge-1 operations as they follow lexical-morphological processes. The input sentence is represented at line 3–6. The processing then begins from the first word *Seine* (line 8–19) which is decomposed into two morphemes D^0 and N^0 (lines 14–19) plus the inflectional features of the former (line 8–11). The stage at which D^0 is in the syntactic working memory and N^0 is merged-1 (§ 4.2) is represented at lines 14–19. Lines 16–18 list the various merge-1 sites and their ranking § 4.2, 5.3.4–5.3.8 and then the site that was chosen (line 19). Since there is only constituent in the working memory when N is merged, the only solution available is $D(N)^0$ which creates a complex head § 4.5 corresponding to *Seine*. When the structure in the working memory is more complex, there are more candidate sites (e.g., lines 43–44). Symbols #0x5# are arbitrary address symbols that allow the user to jump into the corresponding parsing tree in the log file, which can be very long and complex and thus difficult to search. Once all input elements have been consumed, the result is submitted to transfer for interpretation. Figure 21 illustrates transfer § 4.8, thus we see head reconstruction (lines 72, 73), A-chain (line 76),

agreement operations (77-79) and the LF-interface representation (82). The candidate solution, in this case the first-pass parse, is on line 70.

```
70    Transfer [[D Seine] [T(V) [towards D(N)]]] to LF:---------------------------
71
72        Head Chain(D) => [[D Seine] [T(V) [towards [D Paris]]]]
73        Head Chain(T) => [[D Seine] [T [flow [towards [D Paris]]]]]
74        T acquired ϕ-completeness.
75        T triggers A-movement
76        Phrasal Chain(T) => [[D Seine]:3 [T [__:3 [flow [towards [D Paris]]]]]]
77        towards° agrees with D and values PHI:DET:DEF PHI:NUM:SG PHI:PER:3
78        flow° probes own independent pro and found nothing
79        T° agrees with D and values PHI:DET:DEF PHI:NUM:SG PHI:PER:3
80
81        Syntax-semantics interface endpoint:
82        [[D Seine]:3 [T [__:3 [flow [towards [D Paris]]]]]]
83
```

If transfer succeeds, narrow semantic interpretation is triggered and recorded into the log file:

```
81        Syntax-semantics interface endpoint:
82        [[D Seine]:3 [T [__:3 [flow [towards [D Paris]]]]]]
83
84        LF-interface and postsyntactic legibility tests:
85        Interpreting TP globally:
86            Argument for T°: [D Seine] (by second merge).
87            Project [D Seine]: (1, QND)(2, GLOBAL)
88            Argument for flow°: [D Seine] (by second merge).
89            Project predicate 'flow': (3, PRE)(4, GLOBAL)
90            Argument for towards°: pro/[D Paris] (by zero merge).
91            Project [D Paris]: (5, QND)(6, GLOBAL)
92        Denotations:
93            [D Seine]~['2', '6']
94            [D Paris]~['2', '6']
95        Assignments:
96            Assignment {'1': '2', '5': '2'}: Rejected by binding.
97            Assignment {'1': '2', '5': '6'}: Accepted.
98            Assignment {'1': '6', '5': '2'}: Accepted.
99            Assignment {'1': '6', '5': '6'}: Rejected by binding.
100       Calculating information structure...{'Marked topics': [], 'Neutral gradient': ['[D Seine]', '[towards [D Paris]]',
101       Accepted.++
102       Solution accepted at 1027ms stimulus onset.
```

6.4.3 Simple logging

A file ending with *_simple_log.txt* contains a simplified log file which shows only a list of the partial phrase structure representations and accepted solutions generated during the derivation.

6.4.4 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 (§ 6.4.2) and does not appear in this file.

6.4.5 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 (§ 6.3.4). The figure below illustrates the phrase structure representation generated for the clause *Seine flows towards Paris*.

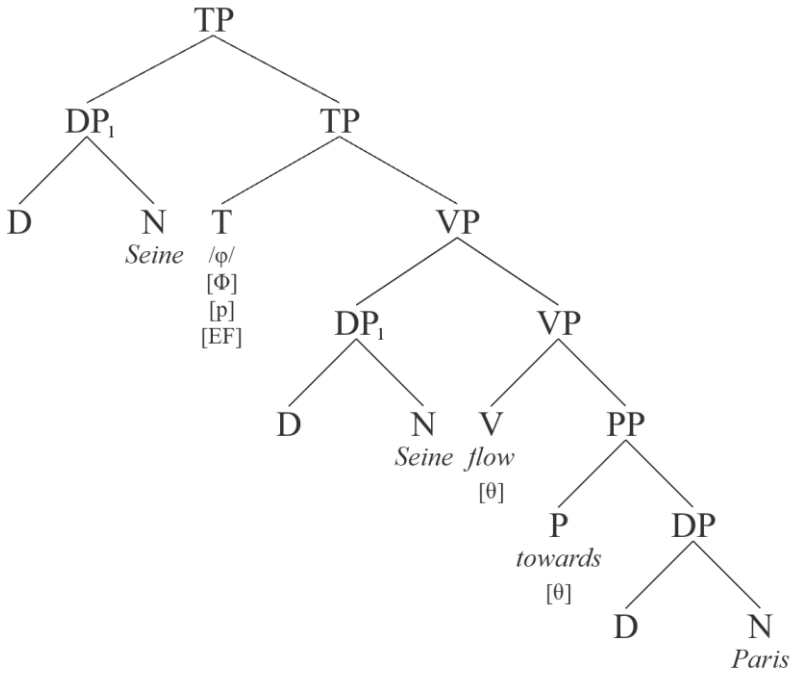


Figure. A simple phrase structure image generated by the algorithm for a simple transitive clause *Seine flows towards Paris*.

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 (later this will be part of the simulation configuration file). If the label of an element is not recognized, it will appear as X (and XP for phrases). The content of these images is controlled by several parameters provided in the *config_study.txt* file. Parameters *show_features* provides a list of lexical features that the user wants to show on the three. For example, if this list contains the feature FIN, then all finite heads

will show [FIN] below them. In some cases the original features that operate inside the implementation code are not elegant or do not match with the names used in the research article; in some cases the original features are so long that they will overlap with other material in the tree. The user can provide abbreviations or substitute symbols, but this mapping can currently be defines only inside the code itself. *Show_words* will show the phonological entries provided for each lexical element. *Spellout* will draw images also for the spellout structures before any transfer operations had takes place. *Case* will add case features to the lexical items. *Show_sentences* will stamp the input sentence into the image. *Show_glosses* will add English translation to each lexical item.

6.4.6 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. The list of resources reported is provided in the parser class and can be modified there. In addition to listing resource consumption, each line also contains the study number (as specified in the input parameters) and the numerical classifications read from the test corpus file, if any (see § 6.3.1).

6.4.7 *Semantic interpretation*

Details of the postsyntactic semantic interpretation are recorded in file *__semantics.txt*, which contains the original sentence, syntactic analysis, summary of semantic interpretation and a detailed listing of the contents of the semantic inventories. The same information can be found also from the derivational log file.

6.4.8 *Control and thematic roles*

Information concerning thematic interpretation and control can be found from the derivational log file, results file and, in a summarized form, from the file *__control.txt*.

6.5 Main script

The script that runs one complete study is called *run_study()*. It is in the *main.py* module. It has one argument *args*, which is a dictionary containing key-value pairs that provide parameters for the simulation. The argument dictionary is created by *__main__.py* function when it reads command line arguments provided by the user. Normally there are no command line arguments, rather, they are provided in the *study_config.txt* file. The script will then prepare I/O operations and configures the simulation.

```
7   def run_study(args):
8       sentence = args.get('sentence', '')
9       local_file_system = LocalFileSystem()
10      local_file_system.initialize(args)
11      local_file_system.configure_logging()
```

Line 10 initializes the simulation by using the parameters provided by the user. All output files are prepared here for writing. Line 11 prepares logging. Next, the script prepares parsers for all input languages. Line 8 handles the situation where the user provides the input sentence as part of the command prompt execution.

```
13      parser_for = {}
14      lang_guesser = LanguageGuesser(local_file_system.external_sources["lexicon_file_name"])
15      for language in lang_guesser.languages:
16          parser_for[language] = LinearPhaseParser(local_file_system, language)
17          parser_for[language].initialize()
```

It checks what languages are present in the lexicon (line 14) and then prepares the speaker model for each language (lines 15–17). A speaker model is an instantiation of the linear phase model (the parser) together with language specific parameters, thus notice that it takes language as an input parameter, which then becomes a "contextual parameter" of the model. It is assumed that a bilingual speaker can change this contextual parameter depending on the language being used; at the level of code, we change the brain model on the basis of the language in the input sentence. The computational core of the model has no language-specific parameters; rather, they came into effect when the functional lexicon is processed through lexical redundancy rules. The issue is empirically nontrivial. Currently, it is assumed that only the functional lexicon changes. When the lexicon is loaded, all lexical items that do not have specification for language – usually these are only functional items – will be provided one by using the contextual language parameter. This will trigger language-specific redundancy rules when these rules are applied.

Next the main script reads all input sentences (if no sentence was given in the input).

```
19  if not sentence:
20      sentences_to_parse = [(sentence, group, part_of_conversation)
21                            for (sentence, group, part_of_conversation)
22                            in local_file_system.read_test_corpus()]
23  else:
24      sentences_to_parse = [([word.strip() for word in sentence.split()], '1', False)]
```

If the user did not provide an input sentence as part of the command prompt executing the simulation (line 18), the test corpus file is used (lines 20–22). The test corpus reader returns tuples that contain the sentence, its experimental group number and whether it is part of a conversation. The user can add any information required to the reader (e.g., contextual tags) and then change the code here to correspond to these changes. These sentences are then forwarded to the parser brain model, one at a time.

```
26      sentence_number = 1
27      for sentence, experimental_group, part_of_conversation in sentences_to_parse:
28          if not is_comment(sentence):
29              language = lang_guesser.guess_language(sentence)
30              local_file_system.print_sentence_to_console(sentence_number, sentence, language)
31              parser_for[language].parse_sentence(sentence_number, sentence)
32              local_file_system.save_output(parser_for[language],
33                                            sentence_number,
34                                            sentence,
35                                            experimental_group,
36                                            part_of_conversation)
37              if not part_of_conversation:
38                  parser_for[language].narrow_semantics.global_cognition.end_conversation()
39              sentence_number = sentence_number + 1
40          else:
41              local_file_system.write_comment_line(sentence)
```

For each sentence this function tries to guess the language, prints the sentence to console, sends it for the parser and saves the output when the parser has finished the job. The code separates comment and sentence processing (line 28). Output generation is handled on line 32 which updates the information to various output files. The derivational log file is generated during processing at the moment each separate operation is executed. Once all sentences have been processed, a few closing tasks are performed.

```
43      # Finish processing
44      if local_file_system.settings['datatake_full']:
45          local_file_system.save_surface_vocabulary(parser_for["LANG:EN"].lexicon.surface_vocabulary)
46      local_file_system.close_all_output_files()
47      if local_file_system.settings['check_output']:
48          local_file_system.check_output()
```

Of particular importance is the output check on lines 47-48 which generates the *error reports* file.

7 Grammar formalization

7.1 Basic grammatical notions (phrase_structure.py)

7.1.1 Introduction

The phrase structure class 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 Lexical items

Lexical items are sets of lexical features. Once they enter syntax, they are associated with a phrase structure node and become *constituents*. We can think of the lexical component as providing feature sets wrapped inside constituents in the syntactic component. Phrase structure nodes define the phrase structure geometry.

7.1.3 Phrase structure geometry

Each constituent α can have the *left daughter constituent* and the *right daughter constituent*. Their *mother* will be α. A constituent is *primitive* if and only if it does not have both the left and right immediate daughter constituents. A constituent is complex if it is not primitive.

```
42      def complex(self):
43          return self.right and self.left
44
45      def primitive(self):
46          return not self.complex()
47
48      def is_left(self):
49          return self.mother and self.mother.left == self
50
51      def is_right(self):
52          return self.mother and self.mother.right == self
```

It follows that a constituent that has zero or one immediate daughter is primitive, creating a three-way classification: complex versus primitive constituents, the latter further divided into terminal (with zero daughters) and nonterminal constituents (with one daughter). Nonterminal nonphrasal constituents represent *complex heads* (§ 4.8.5). Only phrasal constituents are in the domain of phrasal syntactic rules. We can then define several other relations, the definitions are self-evident.

```
64      def bottom(x):
65          return list(x.minimal_search())[-1]
66
67      def top(x):
68          while x.mother:
69              x = x.mother
70          return x
71
72      def grandmother(self):
73          if self.mother.mother:
74              return self.mother.mother
75
76      def aunt(self):
77          if self.mother:
78              return self.mother.sister()
```

7.1.4 Complex heads and affixes

A constituent is a *complex head* if and only if it has the right constituent but not the left constituent. The orphan right constituent holds the internal morpheme. A complex head is a primitive constituent, hence not in the domain of phrasal rules, despite containing a constituent.

```
54      def has_affix(self):
55          return self.right and not self.left
56
57      def get_affix_list(x):
58          lst = [x]
59          while x.right and not x.left:
60              lst.append(x.right)
61              x = x.right
62          return lst
```

The algorithm notates the output as X(Y), meaning that morpheme Y is inside X and hence $[_X$ Y]. We could use some other implementation if there were empirical or theoretical reason to do so while keeping the neutral notation. See § 4.8.5.

7.1.5 Sisters

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

```
109     def geometrical_sister(self):
110         if self.is_left():
111             return self.mother.right
112         return self.mother.left
```

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

```
114     def sister(x):
115         while x.mother:
116             if x.is_left():
117                 if not x.geometrical_sister().adjunct:
118                     return x.geometrical_sister()
119                 else:
120                     x = x.mother
121             if x.is_right():
122                 if not x.adjunct:
123                     return x.geometrical_sister()
124                 else:
125                     return None
```

This definition ignores invisible (adjoined) right constituents. This definition is not optional, because these properties should follow from the core properties of adjuncts. The notion of adjunct is 'forced' into the model and is not implemented in a natural way.

7.1.6 Proper selected complement

A (regular) *complement* is defined as the same relation as sisterhood (§ 7.1.5). A *proper selected complement* of a primitive constituent is its right sister.

```
127     def right_sister(self):
128         if self.sister() and self.sister().is_right():
129             return self.sister()
130
131     def proper_selected_complement(self):
132         if self.primitive():
133             return self.right_sister()
```

7.1.7 Heads (labels), maximal projection, container

The recursive labeling algorithm § 4.5 is provided informally in (65).

(65) 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).

In Python:

```
80  def head(self):
81      if self.primitive():
82          return self
83      if self.left.primitive():
84          return self.left
85      if self.right.adjunct:
86          return self.left.head()
87      if self.right.primitive():
88          return self.right
89      return self.right.head()
```

The notion of head of α allows us to define several other notions, such as *maximum projection* ,

```
98  def max(self):
99      x = self
100     while x.mother and x.mother.head() == self:
101         x = x.mother
102     return x
```

and the notions of *container* and *inside*, where container of α is the head of the mother of α and inside(α, β) determines whether α is a projection from β .

```
91  def inside(self, head):
92      return self.head() == head
93
94  def container(self):
95      if self.mother:
96          return self.mother.head()
```

Maximum projection therefore refers to the highest node dominating α that has the same head as α, and container refers to the head of α's mother.

7.1.8 Minimal search, geometrical minimal search and upstream search

Several operations require that the phrase structure is explored in a pre-determined order. The *minimal search* from α explores the right edge of α in a downstream direction and crawls downwards on the right edge of α. It is defined by creating an iteration over phrase structure. The iterator is defined by the Python *__next__* and *__iter__* functions called when constructing iterators.

```
161     def __iter__(self):
162         self.nn = self
163         return self
164 
165     def __next__(self):
166         if not self.nn:
167             raise StopIteration
168         current = self.nn
169         if self.nn.primitive():
170             self.nn = None
171             return current
172         elif self.nn.head() == self.nn.right.head() or self.nn.left.proper_selected_complement():
173             self.nn = self.nn.right
174         else:
175             self.nn = self.nn.left
176         return current.left
```

Suppose we examine constituent α. Constructing an iterator over α calls *__iter__* which sets the next node = α and returns it. α will be the first item in the sequence. The iterator will then call *__next__*, which returns α if it is primitive and sets the next node = None, which terminates the iteration (first condition inside *__next__*). Suppose α = [A B]. The next node in the iterator will be A (*return current.left*). We must also decide whether the iterators goes inside A or B. It branches into B if and only if either (i) $[_{BP}$ A B] or (ii) [A^0 B] (A^0 = primitive and B is a selected right sister). If neither is true, it branches into A. We can now implement a "bare minimal search" by defining an iteration over α. The minimal search function in the current implementation is slightly more abstract and defined as follows:

```
178     def minimal_search(self, selection_condition=lambda x: True, sustain_condition=lambda x: True):
179         return takewhile(sustain_condition, (const for const in self if selection_condition(const)))
```

It has two additional properties: selection function and sustain condition. The former selects items from the bare minimal search based on the condition given in the input (*if selection_condition(const)*), while the latter can be used to define an intervention or termination that halts the search when the condition becomes true (*sustain_condition*). Both are provided as lambda functions. If no functions are provided, all constituents are returned and the search terminates only when a primitive constituent is reaches as defined in the *__next__* function. *Geometrical minimal search* depends on the phrase structure geometry alone and does not respect labelling and visibility.

```
219     def geometrical_minimal_search(x):
220         search_list = [x]
221         while x.complex() and x.right:
222             search_list.append(x.right)
223             x = x.right
224         return search_list
```

7.1.9 Upward paths

The model makes much use of upward paths. Suppose that α constitutes a probe that triggers the scanning operation. First we define the notion of *upward path*.

```
181     def upward_path(self):
182         upward_path = []
183         x = self.mother
184         while x:
185             if x.left.head() != self:
186                 upward_path.append(x.left)
187             x = x.mother
188         return upward_path
```

We proceed upwards and collect all left constituents into a list, ignoring the starting node. *Edge* contains all elements in the path that are inside the projection from α.

```
193     def edge(self):
194         return list(takewhile(lambda x: x.mother and x.mother.inside(self), self.upward_path()))
```

7.1.10 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. It sets up the mother-of relations for both sisters.

```
15      def __init__(self, left_constituent=None, right_constituent=None):
16          self.left = left_constituent
17          self.right = right_constituent
18          if self.left:
19              self.left.mother = self
20          if self.right:
21              self.right.mother = self
22          self.mother = None
23          self.features = set()
24          self.active_in_syntactic_working_memory = True
25          self.morphology = ''
26          self.internal = False
27          self.adjunct = False
28          self.incorporated = False
29          self.find_me_elsewhere = False
30          self.identity = ''
31          self.rebaptized = False
32          self.stop = False
33          self.nn = None
34          self.x = 0
35          self.y = 0
36          if left_constituent and left_constituent.adjunct and left_constituent.primitive():
37              self.adjunct = True
38              left constituent.adjunct = False
```

Some of the properties listed here are technical and only support the implementation (e.g., x, y are used for drawing phrase structure trees; rebaptized keeps track of chain numbering; identity is for bookkeeping; morphology/internal/incorporated assist in morphological decomposition). The feature *find_me_elsewhere* keeps tracks of copies: when set to True, the element is interpreted as been copied elsewhere by reconstruction.[42]

[42] All constituents, whether primitive or not, can have a set of features. This set is currently used only for lexical items, but was originally generalized so that it still applies to all constituents. A complex head [α β] takes advantage of this option: here [α β] is associated with the lexical features of α, and we interpret

7.1.11 Countercyclic Merge-1

Countercyclic Merge-1 (merge_1(α, β, direction)) targets constituent α inside an existing partial 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]. Thus, if we have a phrase structure [X....α....Y], then Merge-1 generates either (a) or (b).

(66)

a. $[X...[_{\gamma} \alpha\ \beta]...Y]$

b. $[X...[_{\gamma} \beta\ \alpha]...Y]$

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 can occur. The fact that Merge-1 dissolves into separate processes is reflected in the code, which 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.

---

the constituency relation as creating a linear sequence between α and β, thereby "chaining" the two feature bundles together. The system makes room for "complex lexical items" $[\alpha\ \beta]_F$ which would be complex constituents associated with (lexical) feature bundles F.

```
788     def Merge(self, C, direction=''):
789         local_structure = self.local_structure()
790         new_constituent = self.asymmetric_merge(C, direction)
791         new_constituent.substitute(local_structure)
792         return new_constituent
793 
794     def asymmetric_merge(self, B, direction='right'):
795         self.consume_resources('Merge-1', self)
796         if direction == 'left':
797             new_constituent = PhraseStructure(B, self)
798         else:
799             new_constituent = PhraseStructure(self, B)
800         return new_constituent
801 
802     def substitute(self, local_structure):
803         if local_structure.mother:
804             if not local_structure.left:
805                 local_structure.mother.right = self
806             else:
807                 local_structure.mother.left = self
808             self.mother = local_structure.mother
```

7.1.12 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.

```
823     def remove(self):
824         if self.mother:
825             mother = self.mother
826             sister = self.geometrical_sister()
827             grandparent = self.mother.mother
828             sister.mother = sister.mother.mother
829             if mother.is_right():
830                 grandparent.right = sister
831             elif mother.is_left():
832                 grandparent.left = sister
833             self.mother = None
```

7.1.13 Detachment

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

```
868     def detach(self):
869         is_right = self.is_right()
870         original_mother = self.mother
871         self.mother = None
872         return original_mother, is_right
```

7.1.14 Feature checking

There are currently two feature checking operations, one which checks all features listed in the input set and another which checks some features listed in the input set.

```
644     def check(self, feature_set):
645         return feature_set == {'*'} or feature_set & self.head().features == feature_set
646
647     def check_some(self, feature_set):
648         return feature_set == {'*'} or feature_set & self.head().features
```

7.1.15 Probe-goal: probe(label, goal_feature)

Probe-goal relations are interpreted as downward pointing long-distance dependencies that are created by following minimal search. The function is currently used only to implement nonlocal selection. Suppose P is the probe head, G is the goal feature, and α is its (non-adjunct) sister in configuration [P, α], then we have define a probe-goal relation (67).

(67) Probe-goal

> Under [P, α], G the goal feature, search for G from left constituents by going downwards inside α along its right edge by using minimal search.

For everything else than criterial features, G must be found from a primitive head to the left of the right edge node. The present implementation has an intervention clause which blocks further search if search encounters an intervention feature or a special condition is satisfied that involves probing for a tail features. The intervention system is not fully developed, and there is no systematic study concerning its properties.

```
    def probe(self, intervention_features, G):
        if self.sister():
            for node in self.sister().minimal_search():
                if node.check({G}) or (G[:4] == 'TAIL' and G[5:] in node.scan_features('OP')[0]):
                    return True
                if node.check(intervention_features):
                    break
```

The probe-goal mechanism implements nonlocal selection. One example of nonlocal selection is the relationship between D and N. Every D must be paired with an N, but the relationship can be intervened by a number of other functional heads such as Q or Num.

7.1.16 Tail-head relations

A tail-head dependency is formed when a probe P must locate a goal G either inside the head of its own projection (“strong tail test”) or inside an upward path by memory scanning (22)(“weak tail test”). The tail test can be *positive*, in that it checks the existence of G, or *negative*, where it checks the absence of G (the test fails if G is present). Goals G are defined by *target features* F = $\{f_1 \ldots f_n\}$ which must all be checked in order for the dependency to form. The main function is as follows.

```
def tail_test(self):
    pos_tsets = {frozenset(positive_features(tset)) for tset in self.get_tail_sets() if positive_features(tset)}
    neg_tsets = {frozenset(negative_features(tset)) for tset in self.get_tail_sets() if negative_features(tset)}
    checked_pos_tsets = {tset for tset in pos_tsets if self.tail_condition(tset)}
    checked_neg_tsets = {tset for tset in neg_tsets if self.tail_condition(tset)}
    return pos_tsets == checked_pos_tsets and not checked_neg_tsets
```

The function collects positive and negative target features and checks the tail condition for each. The test passes if all positive features are checked but none of the negative features are. The tail condition is defined as follows.

```
def tail_condition(self, tset):
    if not self.referential() and \
            self.max() and \
            self.max().container() and \
            (self.max().container().check(tset) or
             (self.max().mother.sister() and
              self.max().mother.sister().check(tset))):
        return True
    if self.referential() or self.preposition():
        for m in (affix for node in self.upward_path() if node.primitive() for affix in node.get_affix_list()):
            if m.check_some(tset):
                return m.check(tset)
```

The test checks two different condition depending on whether the test its is referential (or preposition) or not. Nonreferential constituents must satisfy a stronger condition that is defined in the first clause and essentially tests that the tail features are present in the head of the projection inside of which the testing element is.

```
741             self.max() and \
742             self.max().container() and \
743             (self.max().container().check(tset) or
744              (self.max().mother.sister() and
745               self.max().mother.sister().check(tset))):
```

This applies to adverbials, which are required to appear inside projections from specific heads. The second clause uses a weaker test and only requires that the tail feature can be found inside an upward path.

```
747         if self.referential() or self.preposition():
748             for m in (affix for node in self.upward_path() if node.primitive() for affix in node.get_affix_list()):
749                 if m.check_some(tset):
750                     return m.check(tset)
```

7.1.17 Abstractions

The phrase structure class contains a large number of abstractions which convert lexical features into technical terms. This allow the researcher to change the definitions in one place. Below a few examples.

```
1087    def adverbial(self):
1088        return self.check({'Adv'})
1089
1090    def nominal(self):
1091        return self.check({'N'})
1092
1093    def light_verb(self):
1094        return self.check_some({'v', 'v*', 'impass', 'cau'})
1095
1096    def force(self):
1097        return self.check({'FORCE'})
```

7.2 Transfer (transfer.py)

7.2.1 A comment on the current version (15.0)

The organization and operation of the transfer function has undergone major refactoring and simplification, therefore this version should be regarded as the first version of a longer evolution. It will be documented here in its incomplete stage due to the fact that the current source code does not match at all with what was documented in previous versions.

Before examining the logic of the code, a few words towards the general philosophy behind the current approach. As discussed in § 4.8, transfer seems to be composed out of several operations, such as head reconstruction, A-chains, Ā-chains and scrambling, that on the surface have different properties. For this reason in many earlier models each reconstruction operation was defined in its own class and utilized functions that were specific to that operation. It was simply impossible to calculate the data without using specialized modules for each operation. On the other hand, it was clear from the beginning that they were based on the same general template when examined from a more abstract point of view. Each operation was triggered by a deviance or error in the input and sought to correct it. An error in this context means a configuration or property in the spelled out structure (the structure that was submitted to transfer) that cannot pass LF-legibility and/or semantic interpretation. In the current version, some significant steps have been taken to unify all reconstruction operations under the same function, but the work is not complete; therefore, what exists at this stage is a two-factored system where all reconstruction operations utilize the same template but a residuum of differences remain that are defined elsewhere.

7.2.2 Generalized transfer

Reconstruction is a reflex-like operation that takes place without interruption from the beginning to the end. From an external point of view, it constitutes one step; internally the operation is a sequence implemented by the function *execute_sequence*:

```
    def execute_sequence(self, ps):
        log(f'\n\tTransfer {ps} to LF:--------------------------------------------
        self.reconstruct(ps, self.instructions['Head'].copy())
        self.reconstruct(ps, self.instructions['Feature'].copy())
        self.reconstruct(ps, self.instructions['Extraposition'].copy())
        self.scrambling_module.reconstruct(ps)
        self.reconstruct(ps, self.instructions['Phrasal'].copy())
        self.reconstruct(ps, self.instructions['Agree'].copy())
        self.last_resort(ps, self.instructions['Last Resort Extraposition'].copy())
        log(f'\n\n\t\tSyntax-semantics interface endpoint:\n\t\t{ps.top()}\n')
        return ps.top()
```

The general form of each step is defined by *reconstruct* which takes two arguments, the starting node of each cycle, usually the bottom right node, and *instructions*. The scrambling module has different form; it has not been unified with the rest of the transfer operations. See 7.2.8. The *instructions* parameter is a dictionary which contains lambda-functions providing the conditions and operations that are observed and executed during each reconstruction cycle, depending on the type of reconstruction. This dictionary, shown below, therefore contains most of the operation-specific information that remains in the current theory and which have not been yet reduced away.

```
self.instructions = {'Head':          {'type': 'Head Chain',
                                       'test integrity': lambda x: x.has_affix() and not x.right.find_me_elsewhere,
                                       'repair': lambda x, y, z: x.create_chain(y, z),
                                       'selection': lambda x: True,
                                       'sustain': lambda x: True,
                                       'legible': lambda x, y: y.properly_selected() and not y.empty_finite_EPP() and y.right_sister() != x},
                     'Phrasal':       {'type': 'Phrasal Chain',
                                       'test integrity': lambda x: x.EF(),
                                       'repair': lambda x, y, z: x.create_chain(y, z),
                                       'selection': lambda x: x.primitive() and not x.finite(),
                                       'sustain': lambda x: not (x.primitive() and x.referential()),
                                       'legible': lambda x, y: x.Abar_legible(y),
                                       'last resort A-chain conditions': lambda x: x == x.container().licensed_phrasal_specifier() or x.VP_for_fronting()},
                     'Feature':       {'type': 'Feature Chain',
                                       'test integrity': lambda x: x.check({'?ARG'}) or x.check({'Fin'}),
                                       'repair': lambda x, y, z: x.feature_inheritance()},
                     'Agree':         {'type': 'Agree',
                                       'test integrity': lambda x: x.is_unvalued(),
                                       'repair': lambda x, y, z: self.Agree_variations.Agree(x, y)},
                     'Extraposition':  {'type': 'Extraposition',
                                       'test integrity': lambda x: (x.top().contains_finiteness() or x.top().referential()) and x.induces_selection_violation() and x.sister() and not x.sister().adjunct,
                                       'repair': lambda x, y, z: x.extrapose(self)},
                     'Last Resort Extraposition': {'type': 'Last Resort Extraposition',
                                                   'test integrity': lambda x: (x.top().contains_finiteness() or x.top().referential()) and not self.brain_model.LF.LF_legibility_test_detached(x.top()),
                                                   'repair': lambda x, y, z: x.last_resort_extrapose(self)}
                     }
```

Crucially, the dictionary contains lambda functions which provide the operation-specific properties. In an ideal word they would not exist; rather, all reconstruction would use the same functions. The reconstruction cycle itself is defined as follows.

```
62      def reconstruct(self, probe, inst):
63          x = probe.bottom()
64          while x:
65              if inst['test integrity'](x):
66                  inst['repair'](x, self, inst)
67              x = x.selector()
```

The function targets the starting node (usually the bottom node), examines if the node needs repair and, if it does, calls the repair function as specified in the instructions. Each *reconstruction cycle* moves from the bottom node towards the top node, moving from head to head, and performs the required repair operations for each node it finds to require repair. The nodes needing repair are detected by the *need repair* entry in the instructions dictionary, a lambda function that is provided by the caller and which again depends on the nature of the cycle. For example, head reconstruction is triggered whenever the node is a complex head.

Because transfer is executed in one well-ordered sequence, as shown above, this loop is executed several times, once for every reconstruction type. An alternative, perhaps the preferred alternative, is to run this loop only once but perform all reconstruction operations at each step. The reason this is currently impossible is the fact that scrambling does not (and cannot) use this mechanism but must be ordered in the exact way illustrated by the execute sequence function.[43]

7.2.3 Chain creation (all chains)

Once the repair function detects that chain formation could be needed, a *create_chain* function in the phrase structure class is called. This function creates head-, A- and Ā-chains, but does not (yet) apply to scrambling chains.

[43] The most likely solution to this problem is to dissolve the scrambling operation into two separate operations, one taking care of the (i) right edge (unified with extraposition) and another which uses the (ii) bottom-up cycle. Then we can execute (i) as part of the extraposition step while (ii) gets unified with the rest of the mechanism.

```
407     def create_chain(self, transfer, inst):
408         for target in self.select_targets_from_edge(inst):
409             inst, target = self.prepare_chain(target, inst, transfer)
410             self.form_chain(target, inst)
411             transfer.brain_model.consume_resources(inst['type'], self)
412 
413             # Successive-cyclicity
414             if target.primitive() and inst['test integrity'](target):
415                 target.create_chain(transfer, inst)
416             elif target.max().container() and inst['test integrity'](target.max().container()):
417                 target.max().container().create_chain(transfer, inst)
```

The function is called by the probe head H (*self*). The iteration at the beginning selects relevant objects from the edge of H: the head itself in the case of head chain, phrasal specifiers in the case of phrasal chains.

```
419     def select_targets_from_edge(self, instructions):
420         if instructions['type'] == 'Phrasal Chain':
421             return [x for x in self.edge() if not x.find_me_elsewhere and not self.licensed_expletive(x)]
422         return [self.right]
```

There can be several such elements, as in the case of Italian left periphery, for example. The selected items are called *targets*. Licensed expletives are ignored. The chain is then prepared (see below) and created by *form_chain* function for the *target* element designated by the iteration. If the reconstructed item needs further chain repair, the same function is called recursively. A complex head that hosts several affixes gets cleaned up completely, since only the last step creates a primitive head. In the case of phrasal targets, the result is successive-cyclic reconstruction. In other words, successive-cyclic reconstruction/movement is produced by the last four lines in this function.

Before the chain is formed, the operation is prepared (function *prepare_chain*). This operation creates a copy of the target element that will be reconstructed, distinguishes A-chains from Ā-chains, and handles feature copying and null head generation in the case of Ā-chains.

```
389     def prepare_chain(probe, specifier, inst, transfer):
390         if inst['type'] == 'Phrasal Chain':
391             if probe.scan_criterial_features(specifier, 'ΔOP'):
392                 if not specifier.supported_by(probe):
393                     probe = specifier.sister().Merge(transfer.access_lexicon.PhraseStructure(), 'left').left
394                     probe.features |= probe.add_scope_information()
395                     log(f'\n\t\tCreated {probe}° ')
396             else:
397                 inst['selection'] = lambda x: x.has_vacant_phrasal_position()
398                 inst['legible'] = lambda x, y: True
399                 log(f'\n\t\t{probe} triggers A-movement')
400             probe.copy_criterial_features(specifier)
401             probe.features = transfer.access_lexicon.apply_redundancy_rules(probe.features)
402         return inst, specifier.copy_for_chain(transfer.babtize())
```

In the case of head chains, only the last line is executed which copies the target head before reconstruction (*copy_for_chain*; the *babtize* function provides chain indexes for the output). If the target element is a phrase, a decision is made between A/Ā-chain generation such that the latter is applied if and only if the reconstructed target phrase is an operator. A-chain is currently a last resort operation implemented by providing the default values for the instructions dictionary. See 7.2.5. If an Ā-chain is generated, then the feature content of the probe head is modified and a new probe head is generated in the case of double specifier structure. Once the chain has been prepared, it is *formed*:

```
432     def form_chain(self, target, inst):
433         for head in self.minimal_search_domain().minimal_search(inst['selection'], inst['sustain']):
434             if head != self and head.test_merge(target, inst['legible'], 'left'):
435                 break
436             target.remove()
437         else:
438             if not self.top().bottom().test_merge(target, inst['legible'], 'right'):
439                 target.remove()
440                 if self.sister():
441                     self.sister().Merge(target, 'left')
```

Chain formation executes a minimal search (7.1.8) by using the selection and sustain parameters provided in the instructions and finds the closest position where the element is legible. If minimal search reaches the end but no suitable position came up, two further operations are attempted: merge to the right of the bottom node and then merge just below the probe head. The last option constitutes a last resort strategy that is always applied if nothing else works.

7.2.4 Head chain

A head chain is triggered when the reconstruction cycle finds a complex head X(Y). Head Y is targeted inside X and moved downstream by minimal search (7.1.8) until a position is found where it can be selected. The chain creation algorithm is called recursively if Y itself is a complex head. Proper selection is defined as follows:

```
314     def properly_selected(self):
315         return self.selector() and self.check_some(self.selector().licensed_complements())
```

Most likely this version does not yet handle long head movement that was part of previous models. It has not been unified with the new architecture.

7.2.5 A-chains

A-chains constitute a last resort option when an operator Ā-chain cannot be created and if the target satisfies conditions for the A-chain. The decision is currently based on whether the reconstructed specifier is an operator or not.

```
391                 if probe.scan_criterial_features(specifier, 'ΔOP'):
392                     if not specifier.supported_by(probe):
393                         probe = specifier.sister().Merge(transfer.access_lexicon.PhraseStructure(), 'left').left
394                         probe.features |= probe.add_scope_information()
395                         log(f'\n\t\tCreated {probe}° ')
396                 else:
397                     inst['selection'] = lambda x: x.has_vacant_phrasal_position()
398                     inst['legible'] = lambda x, y: True
399                     log(f'\n\t\t{probe} triggers A-movement')
```

The target phrase is merged to the first vacant position found by minimal search (7.1.8). A vacant position means either the complement of the bottom right node ([XP$_1$...[Y __$_1$]]) or a position between two heads that does not contain other phrases ([XP$_1$...[Y __$_1$ [Z...]]]).[44]

```
474     def has_vacant_phrasal_position(self):
475         return self.gapless_heads() or self.is_right()
```

[44] The operation is too simple to handle nontrivial A-chain data. Either an additional condition is added which requires that the operations targets only thematic position (licensing long-distance A-chains) or we add successive-cyclicity by relying on the recursion used in connection with head chains.

If the target element does not satisfy the conditions for A-chains, it is not reconstructed. It may be reconstructed by a later operation. Head reconstruction and scrambling reconstruction are applied before A-chains.

7.2.6 Ā-chains

Ā-chains are formed when the target element is an operator, i.e. contains an operator feature. Minimal search (7.1.8) finds the first specifier or complement position where the element can be selected.

```
450    def Abar_legible(self, y):
451        if y == self.next(self.edge):
452            if len(self.edge()) < 2 and self.specifier_match(y) and self.specifier_sister().tail_match(self.specifier_sister(), 'left'):
453                return True
454        if self.sister() == y:
455            return self.complement_match(y)
```

Self is the head H under consideration and *y* is the reconstructed phrase. The first clause handles reconstruction into a specifier position, the latter to the complement position. A specifier reconstruction is accepted if the H accepts y as a specifier, there is only one specifier position (tucking in operation is disabled) and the reconstructed object satisfies the tail test. Complement reconstruction is accepted if H accepts y as its complement.

```
302    def complement_match(self, const):
303        return const.check_some(self.licensed_complements())
```

```
291    def specifier_match(self, phrase):
292        return phrase.head().check_some(self.licensed_specifiers())
```

7.2.7 Agree

Agree targets probe heads with an uninterpretable φ_ (which are by definition predicates) and locates a goal which values its φ-features. In Python:

```
504    def Agree(self):
505        self.value_features_from(self.get_goal())
```

Self represents the probe head, which gets features from the goal selected by the *get_goal()* function. The goal is located by minimal search § 4.8.7, 7.1.8. In Python:

```
508     def get_goal(self):
509         return next(self.minimal_search_domain().minimal_search(lambda x: (x.head().referential() and
510                                                                             not x.find_me_elsewhere) or
511                                                                            x.phase_head(),
512                                                                 lambda x: not x.phase_head()), self)
```

The function *minimal_search_domain()* determines the starting point of the search. For example, if the probe does not have a sister, then the search begins from the probe itself. Minimal search targets the selected starting point and looks for referential heads at their base positions and stops at the phase head. The user can define these properties by changing the lambda functions representing the function parameters. If nothing is found, the function results the probe itself. If a goal is found, its φ-features are *valued* to α.

```
514 def value_features_from(self, goal):
515     log(f'\n\t\tAgree({self}°, {goal.head()}) values ')
516     for phi, phi_ in [(i(phi), self.unvalued_counterparty(i(phi))) for phi in sorted(list(goal.head().features)) if self.target_phi_feature(phi, goal)]:
517         self.value_feature(phi, phi_, goal)
518     self.AgreeLF(goal)
```

The function gets the phi-features from the goal from a predetermined set defined by *target_phi_features*(). They are stored into (*phi*, *phi_*) tuples where phi is the incoming φ-feature and *phi_* is an unvalued counterparty if any, which results in valuation.

```
520     def value_feature(self, phi, phi_, goal):
521         log(f' {phi}')
522         if not self.feature_licensing(phi):
523             self.features.add('*')
524             log(f'(*)')
525         elif phi_:
526             self.features.discard(phi_)
527             self.features.update({phi, phi.split(':')[1], 'dPHI:IDX:' + goal.head().get_id()})
```

First we determine if there is a feature conflict at the probe head that will crash the derivation (lines 524-526). This will capture agreement mismatches. Valuation is performed next (lines 528-529). The function *AgreeLF* in the *value_feature_from()* will update features of the probe if $\text{Agree}_{\text{LF}}(\alpha, X)$ occurred.

```
529     def AgreeLF(self, goal):
530         if self != goal:                        # Definition of AgreeLF
531             self.add_features({'Φ', 'ΦLF'})     # Feature consequences of AgreeLF
532             if self.check({'!SELF:Φ'}):         # Evoke [p]
533                 self.features.add('!SELF:p')
```

7.2.8 Scrambling (scrambling_reconstruction.py)

Scrambling reconstructions has not yet been unified with the generalized reconstruction function. It performs symmetric minimal search from the top of the structure, detects adjuncts and reconstructs them.

```
16  def reconstruct(self, ps):
17      for target in ps.symmetric_minimal_search(lambda x: x.trigger_adjunct_reconstruction(), lambda x: x.is_right()):
18          self.reconstruct_scrambled_item(target)
```

Symmetric minimal search returns both A and B under [A B].

```
226 def symmetric_minimal_search(self, condition=lambda x: x == x, stop_condition=lambda x: x == x):
227     lst = []
228     for node in self.top().minimal_search():
229         if condition(node):
230             lst.append(node)
231             if stop_condition(node):
232                 break
233         if node.sister() and condition(node.sister()):
234             lst.append(node.sister())
235             if stop_condition(node.sister()):
236                 break
237     return lst
```

Reconstruction is implemented by the following function.

```
20  def reconstruct_scrambled_item(self, target):
21      if target.is_right():
22          self.adjunct_constructor.externalize_structure(target.head())
23          if target.legible_adjunct() or target.head().adverbial() or not target.top().contains_finiteness():
24              return
25
26          starting_point = target.container()
27          if target.is_left():
28              starting_point.head().features.add('p')
29          virtual_test_item = target.copy()
30          local_tense_edge = target.local_tense_edge()
31          for node in local_tense_edge.minimal_search(lambda x: x == x, lambda x: self.sustain_condition(x, target, local_tense_edge)):
32              self.merge_floater(node, virtual_test_item)
33              if self.test_adjunction_solution(node, target, virtual_test_item, starting_point, 'left'):
34                  break
35          else:
36              node.Merge(virtual_test_item, 'right')
37              virtual_test_item.adjunct = False
38              self.test_adjunction_solution(node, target, virtual_test_item, starting_point, 'right')
```

The function locates the local tense edge and performs a minimal search, trying to find a legitimate position for the scrambled item.

It is clear that these operations are not correct, although they reproduce the desired behavior. The symmetric minimal search does not likely exist, but dissolves into two operations, one which handles the right edge together with extraposition and the second which will be unified with the general bottom-up reconstruction cycle. These properties will be added at some later point, but they require extensive testing.

7.2.9 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?

```
7    def externalize_structure(self, ps):
8        if ps and ps.head().is_adjoinable() and ps.mother:
9            if ps.complex():
10               self.externalize_and_transfer(ps)
11           else:
12               self.externalize_head(ps)
```

If the externalized element was a head, then we need to make a decision on whether its specifier should be taken as well. The decision is made as follows:

```
14   def externalize_head(self, probe):
15       if probe.isolated_preposition():
16           self.externalize_and_transfer(probe)
17           return
18       if probe.externalize_with_specifier():
19           self.externalize_and_transfer(probe.mother.mother)
20       else:
21           self.externalize_and_transfer(probe.mother)
```

Specifier is carried if it is required by an EPP-feature or the special “capture specifier rule” applies, which is expressed by the latter function. The externalization function pulls the phrase into the secondary working space, adds tails features if needed, and transfers it:

```
23      def externalize_and_transfer(self, ps):
24          if ps.mother:
25              ps.adjunct = True
26              self.add_tail_features_if_missing(ps)
27              self.transfer_adjunct(ps)
```

The transfer adjunct function detaches the adjunct temporarily from the structure before transfer is applied.

7.3 LF-legibility (LF.py)

7.3.1 Overall

The purpose of the LF-legibility test is to check that output of the syntactic pathway satisfies the LF-interface conditions and can therefore be interpreted semantically, at least in principle. Only primitive heads will be checked. The test consists of several independent tests, which are collected into a separate data structure as functions.

```
10          self.LF_legibility_tests = [('Selection test', self.selection_test),
11                                      ('Projection Principle', PhraseStructure.projection_principle_failure),
12                                      ('Head Integrity test', PhraseStructure.unrecognized_label),
13                                      ('Feature Conflict test', PhraseStructure.feature_conflict),
14                                      ('Probe-Goal test', PhraseStructure.probe_goal_test),
15                                      ('Semantic Complement test', PhraseStructure.semantic_complement),
16                                      ('Double Specifier Filter', PhraseStructure.double_spec_filter),
17                                      ('Criterial Feature test', PhraseStructure.legitimate_criterial_feature),
18                                      ('Adjunct Interpretation test', PhraseStructure.interpretable_adjunct),
19                                      ('Edge feature test', PhraseStructure.edge_feature_test)]
```

The LF-legibility test function serves as a “gateway” which regulates the tests that are selected for active use, and then calls the recursive test function.

```
32      def LF_legibility_test(self, ps, test_battery=None):
33          if test_battery:
34              self.active_test_battery = test_battery
35          else:
36              self.active_test_battery = self.LF_legibility_tests
37          return self.pass_LF_legibility(ps)
```

If no test battery is given as an argument, then all tests will be used; if a test battery is provided, it will be used. The recursive test function explores all primitive lexical items recursively in the output representation and applies all active tests to it.

```
39      def pass_LF_legibility(self, ps):
40          if not ps.find_me_elsewhere:
41              if ps.primitive():
42                  for (test_name, test_failure) in self.active_test_battery:
43                      if test_failure(ps):
44                          log(f'\n\t\t{ps} failed {test_name} ')
45                          self.error_report_for_external_callers = f'{ps} failed {test_name}'
46                          return False
47              else:
48                  if not self.pass_LF_legibility(ps.left):
49                      return False
50                  if not self.pass_LF_legibility(ps.right):
51                      return False
52          return True
```

Of special interest is the selection test, which examines if the specifier and complement selection tests are valid. Selection test functions are provided in a separate data structure of this class.

```
21          self.selection_violation_test = {'!1EDG': PhraseStructure.selection__negative_one_edge,
22                                           '-SPEC': PhraseStructure.selection__negative_specifier,
23                                           '-COMP': PhraseStructure.selection__negative_complement,
24                                           '!COMP': PhraseStructure.selection__positive_obligatory_complement,
25                                           '!SELF': PhraseStructure.selection__positive_self_selection,
26                                           '-SELF': PhraseStructure.selection__negative_self_selection,
27                                           '-ΦPF':  PhraseStructure.selection__phonological_AGREE}
```

They are applied to any lexical item with the gateway feature (left feature in the above dictionary).

```
54      def selection_test(self, probe):
55          for selected_feature in sorted(for_lf_interface(probe.features)):
56              if selected_feature[:5] in self.selection_violation_test.keys() and \
57                      not self.selection_violation_test[selected_feature[:5]](probe, selected_feature[6:]):
58                  log(f'\n\t\t{probe} failed feature {selected_feature} ')
59                  return True
```

The function activates a selection test if the lexical item has the gate feature, as listed in the selection test list above. If the test fails (*not test(probe, lexical_feature)*).

7.3.2 Selection tests

The selection rules, which are self-explanatory, are shown below.

```
260     # Feature -SPEC:L
261     def selection__negative_specifier(self,  selected_feature):
262         return not self.next(self.edge, lambda x: x.check({selected_feature}) and not x.adjunct)
263
264     # Feature !1EDGE
265     def selection__negative_one_edge(self, selected_feature):
266         return len(self.edge()) < 2
267
268     # Feature !COMP:L
269     def selection__positive_obligatory_complement(self, selected_feature):
270         return self.selected_sister() and self.selected_sister().check({selected_feature})
271
272     # Feature -COMP:L
273     def selection__negative_complement(self, selected_feature):
274         return not (self.proper_selected_complement() and self.proper_selected_complement().check({selected_feature}))
275
276     # Feature [!SELF]
277     def selection__positive_self_selection(self, selected_feature):
278         return self.check({selected_feature})
279
280     # Feature [-SELF]
281     def selection__negative_self_selection(self, selected_feature):
282         return not self.check({selected_feature})
```

7.3.3 Projection principle

Projection principle examines that all referential arguments receive a thematic role. The function first examines if the projection principle applies to the head α, and then verifies that the αP is contained inside a β which assigns it a thematic role.

```
351     def projection_principle_failure(self):
352         return self.max().projection_principle_applies() and \
353                not self.max().container_assigns_theta_role()
```

The latter function is defined as follows:

```
362     def container_assigns_theta_role(self):
363         assigner = self.max().container()
364         return assigner and \
365                (assigner.sister() == self or (self.referential() and assigner.geometrical_sister() == self) or
366                 self.is_licensed_specifier() and assigner.specifier_theta_role_assigner())
```

The first condition (line 365) says that theta roles can be assigned via sisterhood {α, β}. The second condition (line 366) examines the specifier-head configuration [$_{\beta P}$ ... αP [...β...].

7.4 Semantics (narrow_semantics.py)

7.4.1 Introduction

Semantic interpretation is implemented in the module *narrow_semantics.py* which bleeds the syntax-semantics interface (LF-interface). It begins by recursing through the whole structure and interpreting all lexical elements that have content understood by various semantic submodules.

```
126     def interpret_(self, ps):
127         if not ps.find_me_elsewhere:
128             if ps.primitive():
129                 if self.brain_model.local_file_system.settings['calculate_thematic_roles'] and ps.theta_assigner():
130                     thematic_assignment = self.thematic_roles_module.reconstruct(ps)
131                     if thematic_assignment:
132                         self.semantic_interpretation['Thematic roles'].append(thematic_assignment)
133                 if self.brain_model.local_file_system.settings['calculate_predicates'] and ps.check({'ARG'}):
134                     self.semantic_interpretation['Predicates'].append(self.predicates.reconstruct(ps))
135                 self.quantifiers_numerals_denotations_module.detect_phi_conflicts(ps)
136                 self.interpret_tail_features(ps)
137                 if self.brain_model.local_file_system.settings['project_objects']:
138                     self.inventory_projection(ps)
139                 self.operator_variable_module.bind_operator(ps, self.semantic_interpretation)
140                 self.pragmatic_pathway.interpret_discourse_features(ps, self.semantic_interpretation)
141                 if self.failure():
142                     return
143             else:
144                 self.interpret_(ps.left)
145                 self.interpret_(ps.right)
```

All primitive lexical elements are targeted for interpretation; complex phrases are recursed. The general idea is that lexical features are diverged into different semantic subsystems for interpretation.

7.4.2 Projecting semantic inventories (semantic switchboard)

One function of narrow semantics is to project semantic objects, corresponding to the expressions in the input sentence, into the semantic inventories. This is done by function *inventory_projection()*. Suppose we are examining lexical item α. If α has lexical features that can be interpreted by one or several of the semantic subsystems, narrow semantics queries the corresponding system and, if that system can process that lexical feature, asks it to project the corresponding semantic object into existence and then links these objects to the original expression by using a lexical referential index feature. The referential index feature can be thought

of as a link between the expression and the corresponding semantic object. It has form [IDX:N,S] where N is a numerical identifier and S denotes the semantic space.

```
147     def inventory_projection(self, ps):
148         def preconditions(x):
149             return not self.brain_model.first_solution_found and \
150                    not ps.find_me_elsewhere and \
151                    (x.referential() or
152                     (not x.referential() and not x.get_dPHI()))
153
154         if preconditions(ps):
155             for space in self.semantic_spaces:
156                 if self.query[space]['Accept'](ps.head()):
157                     idx = str(self.global_cognition.consume_index())
158                     ps.head().features.add('IDX:' + idx + ',' + space)
159                     log('\n\t\t\tProject ')
160                     self.query[space]['Project'](ps, idx)
161                     self.query[space]['Denotation'] = self.query['GLOBAL']['Project'](ps, self.transform_for_global(self.query[space]['Get'](idx)))
162
163                     # For heuristic purposes so that referential arguments are recognized by BT
164                     if space == 'QND':
165                         ps.head().features.add('REF')
```

Technically the query operation is implemented by using a router data structure *query* that channels lexical instructions to the subsystems that can process them. For example, command

```
if self.query[space]['Accept'](ps.head())
```

sends the instruction 'Accept' plus the head α to the semantic system given by the space parameter, which returns *True* if that system can accept and process α. The command

```
self.query[space]['Project'](ps, idx)
```

then asks the subsystem to project the corresponding element to the semantic inventory. Properties of the projected item are determined by the lexical features in α. Corresponding projection to the global inventory space will also take place.[45] The query data structure itself can be considered like a "switchboard" that is implemented as a dictionary of dictionaries, where the first level dictionary hosts the semantic space identifiers and the second the commands, and where the command is the key and the corresponding implementation function the value. The idea is that

[45] The assumption that every referential expression projects an entity into the discourse inventory might sound odd, since in many cases they should refer to an existing object instead. For example, pronoun *he* will typically denote an existing object. Indeed, it might. However, each time a referential expression is encountered in the input a new object is always projected, as shown by the code above, regardless of whether it will in the end be selected as a likely denotation

narrow semantics functions as a router that diverges the processing of lexical features to various cognitive subsystems.

7.4.3 Quantifiers-numerals-denotations module

The quantifiers-numerals-denotations (QND) module is specialized in processing referential expressions that involve "things" that can be quantified and counted. It projects semantic objects into its own semantic inventory that get linked with referential expressions occurring in phrase structure objects at the syntax-semantics interface. It can also understand and interpret referential lexical features such as φ-features and translate them into semantic features.

The most important function of this module is the calculation of possible denotations and assignments. An assignment is intuitively a mapping between referential expressions in the sentence and denotations in the global discourse space. Assignment generation is performed by the following function:

```
33  def reconstruct_assignments(self, ps):
34      if self.narrow_semantics.brain_model.first_solution_found:
35          self.narrow_semantics.semantic_interpretation['Assignments'] = []
36          return
37
38      log(f'\n\t\tDenotations:')
39      self.referential_constituents_feed = self.calculate_possible_denotations_(ps)
40
41      log(f'\n\t\tAssignments: ')
42      if not self.referential_constituents_feed:
43          return
44      self.create_assignments_from_denotations_(0, 0, {})
45      self.narrow_semantics.semantic_interpretation['Assignments'] = self.all_assignments
```

Line 39 calculates possible denotations for all relevant referential expressions in α and returns a list of expressions [EXP_1, EXP_2, ...] for which this operation was performed. The list is used to make subsequent processing easier and has no cognitive role in the theory. The denotations are stored as denotation sets inside QND space semantic objects (that is, inside this module itself). Denotation sets contain pointers to objects in the global discourse space. For example, if the original expression is pronoun *he*, then the QND space entry may contain a set of denotations {1, 2}, where the numbers refer to two persons 'John$_1$' and 'Simon$_2$' in the global discourse inventory. Once this is done, the function calls recursive assignment function

create_assignments_from_denotations. All logically possible assignments are considered. The assignments are stored into a QND-internal data structure.

Possible denotations are calculated by the following function.

```
47      def calculate_possible_denotations_(self, ps):
48          L1 = []
49          L2 = []
50          if not ps.find_me_elsewhere:
51              if ps.complex():
52                  L1 = self.calculate_possible_denotations_(ps.left)
53                  L2 = self.calculate_possible_denotations_(ps.right)
54              else:
55                  if self.narrow_semantics.has_referential_index(ps, 'QND'):
56                      idx, space = self.narrow_semantics.get_referential_index_tuple(ps, 'QND')
57                      self.inventory[idx]['Denotations'] = self.create_all_denotations(ps)
58                      log(f'\n\t\t\t{self.inventory[idx]["Reference"]}~{self.inventory[idx]["Denotations"]} ')
59                      return [(idx, f'{ps.illustrate()}', ps, self.inventory[idx]['Denotations'])]
60          return L1 + L2
```

The first parts are involved with recursion. We can look at the *else*-clause. A lexical item that has a referential feature linking it with a semantic object in the QND space (*has_referential_index()*) will be provided with denotations by *create_all_denotations()*. The code returns a list of tuples ⟨idx, constituent printout, constituent, denotations⟩, so that the whole recursion will return a longer list containing all expressions that were provided with this information. This list, which is a simple auxiliary representation that plays no role in the theory, will then be used to build the assignments. Function create_all_denotations() will return a set of denotations (global inventory objects) that satisfy the criteria stored in the QND entry. For example, a pronoun *he* can only pick singular male objects, and so on.

```
138     def create_all_denotations(self, ps):
139         return self.narrow_semantics.global_cognition.get_compatible_objects(self.inventory[self.narrow_semantics.get_referential_index(ps, 'QND')])
```

The function *get_compatible_objects()* is part of global cognition, takes semantic criteria as input, and returns a list of all objects in the global discourse space that are compatible with those criteria. Intuitively, here we pick ‘John’ and ‘Simon’ when the criteria are ‘singular, masculine, third party, person’, and things like ‘Mary’ and ‘the horse’ are ignored. Once every referential

expression is associated with a set of denotations, we can generate assignments. This is done recursively:

```
62      def create_assignments_from_denotations_(self, c_index, d_index, one_complete_assignment):
63          idx, const, ps, denotations = self.referential_constituents_feed[c_index]
64          denotation = denotations[d_index]
65          one_complete_assignment[idx] = denotation
66          if len(one_complete_assignment) == len(self.referential_constituents_feed):
67              self.all_assignments.append(self.calculate_assignment_weight(one_complete_assignment))
68          if c_index < len(self.referential_constituents_feed) - 1:
69              self.create_assignments_from_denotations_(c_index + 1, 0, one_complete_assignment.copy())
70          if d_index < len(denotations) - 1:
71              self.create_assignments_from_denotations_(c_index, d_index + 1, one_complete_assignment.copy())
```

We go through all expressions in the list of expressions created by the function that calculated the denotations. The position in that list is given by *c_index*. The we examine all denotations assigned to each such expressions, which is represented by *d_index*. Once every expression has been provided with an assignment, the result is provided with *weight* and stored. The last if-clauses implement recursion (over *c_index* and *d_index*), so that we examine all possible ways of assigning values to the referential expressions. The interesting part is weight calculations:

```
73      def calculate_assignment_weight(self, complete_assignment):
74          weighted_assignment = complete_assignment.copy()
75          log(f'\n\t\t\tAssignment {complete_assignment}: ')
76          weighted_assignment['weight'] = 1
77          for expression in self.referential_constituents_feed:
78              if not self.binding_theory_conditions(expression, complete_assignment):
79                  weighted_assignment['weight'] = 0
80                  log('Rejected by binding.')
81              if not self.predication_theory_conditions(expression, complete_assignment):
82                  weighted_assignment['weight'] = 0
83                  log('Rejected by predication theory.')
84          if weighted_assignment['weight'] > 0:
85              log('Accepted.')
86          return weighted_assignment
```

This function applies the binding theory and predication theory for each complete assignment and drops the weight to zero if a condition is violated. Several logically possible assignments are not considered as possible or likely.

Assignments are first filtered by conditions that mimic the binding conditions A-C, function *binding_theory_conditions()* above. The general idea is that referential expressions can contain grammaticalized features that function as “instructions” for a system that knocks out logically

possible assignments. This filtering is performed by the following function, which in effect incorporates the binding theory.

```
114     def binding_theory_conditions(self, expression, complete_assignment):
115         idx, name, ps, denotations = expression
116         for feature in list(self.get_R_features(ps)):
117             D, rule, intervention_feature, interface = self.open_R_feature(feature)
118             if {rule} & {'NEW', 'OLD'}:
119                 reference_set = self.reference_set(ps, intervention_feature, complete_assignment)
120                 if not self.narrow_semantics.global_cognition.general_evaluation(complete_assignment[idx], rule, reference_set):
121                     return False
122         return True
```

The first lines interpret and handle the lexical R-features that provide instructions to the assignment filter. The main operations are the calculation of the reference set and general evaluation. The reference set is a set of semantic objects that can be accessed from the phrase structure at the syntax-semantics interface by using the particular assignment that is being evaluated and the expression α that is targeted. We will exclude and include assignments based on what is in the reference set. General evaluation will then evaluate, by using instructions provided by the R-feature and the reference set, whether the assignment for the current expression α is possible. These assumptions deduce the effects of binding conditions A-C. The reference set is defined as follows:

```
124     def reference_set(self, ps, intervention_feature, complete_assignment):
125         reference_set = set()
126         for const in (node for node in ps.upward_path() if
127                       self.narrow_semantics.has_referential_index(node.head()) and
128                       self.narrow_semantics.exists(node.head(), 'QND') and
129                       node.head() != ps and
130                       not node.find_me_elsewhere):
131             reference_set.add(complete_assignment[self.narrow_semantics.get_referential_index(const.head(), 'QND')])
132             if intervention_feature and \
133                     not const.find_me_elsewhere and \
134                     {intervention_feature}.issubset(const.head().features):
135                 break
136         return reference_set
```

We pick up all semantic objects accessed by heads inside a constituent vector (upward path, see § 7.1.9) from α. General evaluation, which is part of global cognition, is provided by the following operation that performs a simple set-theoretical comparisons. The intuition is that narrow semantics has direct access to the syntax-semantic interface objects and to the general evaluation operation.

To illustrate these operations by using a concrete example, consider the processing of a simple pronoun *he* that is inside a larger expression α = '. . . *he* . . .'. All referential expressions in α, including the pronoun, are first linked with a set of possible denotations. These sets depend on what entities exists in the global discourse inventory at the time the operation takes place, hence the process takes place at the language-cognition interface. Suppose we have three male persons John$_1$, Simon$_2$ and an unknown third person$_3$ that was projected by default when *he* was first interpreted in the global discourse inventory. The pronoun will be associated with the set {1, 2, 3}, because it could refer to any of these three entities. The system will then consider all possible assignments and select the ones that are most likely and/or plausible, given the context and other factors. Binding theory restricts these assignments. Assignments like *Simon*$_2$ *admires him*$_2$ and *Simon*$_3$ *admires him*$_3$ are both ruled out, because the pronoun would have "too local" antecedent.

7.4.4 Operator-variable interpretation (SEM_operators_variables.py)

The operator-variable module interprets operator-variable constructions. The kernel of the module is constituted by a function that binds operators [OP:F] with the finite propositional scope marker(s) {OP:F, FIN}. The follow function calculates operator bindings:

```
22    def bind_operator(self, head, semantic_interpretation):
23        for operator_feature in (f for f in head.features if not self.scope_marker(head) and self.is_operator_feature(f)):
24            binding = self.interpret_covert_scope(self.find_overt_scope(head, operator_feature))
25            self.interpret_operator_variable_chain(binding, operator_feature, semantic_interpretation)
```

Biding is determined by the result of overt scope computations and covert scope computations, which is then interpreted. The variable *binding* is a dictionary which contains information about the binding dependency. Overt scope computations are defined by

```
28    def find_overt_scope(head, operator_feature):
29        return next(({'Head': head, 'Scope': scope, 'Overt': True} for scope in head.upward_path() if
30                     {operator_feature, 'Fin'}.issubset(scope.features)), {'Head': head, 'Scope': None, 'Overt': False})
```

which looks for a pair of operator and finiteness inside the working memory path. Covert scope is defined by

```
def interpret_covert_scope(binding):
    if not binding['Scope'] and '!SCOPE' not in binding['Head'].features:
        return next(({'Head': binding['Head'], 'Scope': scope, 'Overt': False}
                     for scope in binding['Head'].upward_path() if
                     scope.finite_left_periphery()),
                    {'Head': binding['Head'], 'Scope': None, 'Overt': False})
    return binding
```

which is activated only if overt scope computations have not produced results. It binds the operator to the local finite T or C.

7.4.5 Pragmatic pathway

The pragmatic pathway or module is involved in inferring and computing semantic information that we intuitively associate with (narrow) pragmatics. It has to do with propositional relations between thinkers (speaker, hearer) and propositions and their underlying communicative intentions. The system operates in two ways. On one hand it uses the incoming linguistic information and the context as a source material to infer pragmatic information, such as what is the topic and focus, and what type of communicative moves are involved. These inferential operations run silently in the background and do not change or alter the course of processing inside the syntactic pathway. The pragmatic pathway accesses the information through syntax-pragmatics interfaces. Secondly, the language system and the lexicon can grammaticalize features that activate operations inside the pragmatic pathway in a more direct way, which creates situations where grammatical devices (suffixes, words, prosody, word order, heads, lexical features) affect the pragmatic interpretation in a more direct way. This second mechanism operates through narrow syntax which routes discourse features to the pragmatic system for interpretation. Because these discourse features exist inside the syntactic pathway, they may affect syntactic processing as well.

Let us consider grammaticalized discourse features first. Narrow semantics has a function that directs discourse features to the pragmatic pathway for processing. This function will handle all discourse features.

```
45      def interpret_discourse_features(self, ps, semantic_interpretation):
46          self.refresh_inventory(ps)
47          d_features = self.get_discourse_features(ps.features)
48          if d_features:
49              log('\n\t\t\tInterpreting ')
50          for f in sorted(d_features):
51              log(f'[{f}] at {ps.illustrate()}...')
52              result = self.interpret_discourse_feature(f, ps)
53              if not result:
54                  return []
55              semantic_interpretation['DIS-features'].append(result)
```

The operation sends each discourse feature for interpretation and then stores the results for later use. In the current version, discourse feature interpretation is merely registered in the output.

Calculations involved with the information structure are more fully developed. The operation is called during global semantic interpretation.

```
60      def calculate_information_structure(self, root_node, semantic_interpretation):
61          if root_node.finite():
62              log('\n\t\tCalculating information structure...')
63              semantic_interpretation['Information structure'] = self.create_topic_gradient(self.collect_arguments(root_node))
64              log(f'{semantic_interpretation["Information structure"]}')
65              self.compute_speaker_attitude(root_node)
```

First, the system determines what the root proposition is and what arguments we need to include into the information structural calculations. We are only interested in the thinker (speaker), proposition and the propositional attitude between the two. The system uses this information to calculate a *topic gradient*, which expressed what it thinks constitutes new and old information in the sentence being processed. The system works as follows. When new expressions are streamed into syntax, they are allocated attentional resources by the pragmatic system in the order they appear. This constitutes a syntax-pragmatics interface: it sends information from the syntactic pathway to the pragmatic system. It is registered and processed in the pragmatic system:

```
133     def allocate_attention(self, head):
134         if head.referential() or head.preposition():
135             idx = self.consume_index()
136             head.features.add('*IDX:'+str(idx))
137             self.records_of_attentional_processing[str(idx)] = {'Order':idx, 'Name': f'{head}'}
```

The first line determines what kind of elements are included into the attentional mechanism, and depends on the interests of the researcher. Then, some attentional resources are allocated to the

processing, and order information is stored. Finally, when the sentence comes through the LF-interface, the pragmatic module attempts to construct the topic gradient on the basis of this and other sources of information. The presentation order at which linguistic information was originally presented and processed, as shown above, is now interpreted as representing relative topicality.[46]

Topic gradient calculations are nontrivial for several reasons. First, they must ignore some noncanonical word order changes, such as those created by Ā dependencies. An interrogative direct object pronoun that occurs at the beginning of the sentence should not be interpreted as the topic. Second, the more noncanonical the position is, the more prominent the topic/focus interpretation tends to be. This is especially clear in Finnish. Third, this language allows one the topicalize/focus several constituents (multi-topic/focus constructions) and to perform sentence-internal topicalization/focusing.

Let us consider then the function that constructs the actual topic gradient when the whole sentence is interpreted. It takes the "constituents in information structure" as an argument, which lists the arguments that are part of the proposition the speaker has established a propositional attitude relation. The data structure topic_gradient is then built, which sorts the arguments/semantic objects under consideration and all information about them as recorded by the pragmatic module, and as ordered by their appearance in the comprehension process.

[46] It follows from this that noncanonical word orders can change the way expressions and the semantic objects are represented in the topic gradient. For example, in a language like Finnish with a relatively free word order, various word orders are correlated with distinct information structural interpretations, in fact to such an extent that some movement operations are called "topicalization" and "focussing." This derives the discourse-configurationality profile.

```
def create_topic_gradient(self, constituents_in_information_structure):
    marked_topic_lst = []
    topic_lst = []
    marked_focus_lst = []
    topic_gradient = {key: val for key, val in sorted(self.records_of_attentional_processing.items(), key=lambda ele: ele[0])}
    for key in topic_gradient:
        if topic_gradient[key]['Constituent'] in constituents_in_information_structure:
            if 'Marked gradient' in topic_gradient[key]:
                if topic_gradient[key]['Marked gradient'] == 'High':
                    marked_topic_lst.append(topic_gradient[key]['Name'])
                elif topic_gradient[key]['Marked gradient'] == 'Low':
                    marked_focus_lst.append(topic_gradient[key]['Name'])
            else:
                topic_lst.append(topic_gradient[key]['Name'])
    return {'Marked topics': marked_topic_lst, 'Neutral gradient': topic_lst, 'Marked focus': marked_focus_lst}
```

Next, the elements in the topic gradient are sorted into three lists "marked topics," "neutral gradient" and "marked focus," again in the order they appear in the topic_gradient data structure. The distribution is triggered by information that was stored in connection with the corresponding objects, here during transfer. Thus, in the function implementing adjunct reconstruction there is a function call which registers unexpected word orders used later in the creation of the three-tiered topic gradient. This creates another syntax-pragmatics interface mechanism. How this is done, and where this interface should be positioned in the general architecture, turned out to be extremely nontrivial problem and must be examined in the light of empirical data. The marked topic list, neutral gradient and marked focus lists then appear in the results of the simulation.

7.4.6 Argument-predicate pairs

Every predicate head α must be paired with an argument (or, in some cases, the argument will be assigned by default). The argument must, furthermore, be at the edge of the predicate α at the LF-interface, where "edge" refers to referential heads that are merged "around" α: (i) features of α (pro), then (ii) the complement [α, XP], then (iii) the second-merge specifier [$_{\alpha P}$ XP [α YP]] and, if all these fail, (iv) an element from the extended edge which allows examination of objects merged outside of αP but part of α's upward path. The latter option results in what is usually referred to as finite or nonfinite control in the literature. The edge is searched for an argument in this specific order. The referential head cannot be *inside* any of these constituent. Since these assumptions are experimental and have not yet been published at the present writing, the

implementation is written in a fairly explicit way. First, the notion of edge is defined by enumeration as follows:

```
        self.edge = [('zero merge',
                      lambda x: x.extract_pro() and not x.phi_needs_valuation(),
                      lambda x: x.extract_pro()),
                     ('first merge',
                      lambda x: x.sister() and x.sister().referential(),
                      lambda x: x.sister()),
                     ('second merge',
                      lambda x: x.edge(),
                      lambda x: x.edge()[0]),
                     ('N merge',
                      lambda x: x.phi_needs_valuation(),
                      lambda x: x.control())]
```

The first item in these tuples is the name for the edge position, where "zero merge" refers to the insertion of features inside the predicate head α. The second item is the function which must be satisfied in order for this solution to be considered, and the third is the function which acquired the argument. For example, in order for the complement XP to be selected as an argument for α under [α XP], the complement must exist and it must be referential. The notion of edge is defined by enumeration because the idea is currently at the stage of experimentation; if this solution works, something more general will substitute it. Thus, notice that the edge is actually composed out of several existing grammatical primitives (pro, complement, specifier, control, itself based on upward path).[47] Argument reconstruction is then defined as follows:

```
    def reconstruct(self, probe):
        for name, condition, acquisition in self.edge:
            if condition(probe):
                goal = acquisition(probe)
                if goal:
                    log(f'\n\t\t\tArgument for {probe}°: {self.print_target(probe, goal)} (by {name}).')
                    return f'{probe}°: {self.print_target(probe, goal)}'
        log(f'\n\t\t\t*{probe} was not linked with an argument.')
        self.operation_failed = True
```

[47] Most likely there is just one grammatical notion – say *extended edge* – which includes (i–iv) and which will then be used to define the primitive out of which it is now composed.

We go through all the options defined in the data structure above (line 22), examine the condition (second item, line 23) and, if it is satisfied, try to acquire the argument (line 24). If the argument is found, the iteration stops and the target is returned. If nothing is found, the reconstruction is marked as failure (line 29).

8 Formalization of the parser

8.1 Linear phase parser (linear_phase_parser.py)

8.1.1 The speaker model

The linear phase (LP) parser module *linear_phase_parser.py* defines the behavior of the parser. It defines an idealized speaker model for a speaker of some language. Languages and their speakers differ from each other. These differences are represented in the lexicon, most importantly in the functional lexicon. Each time the linear phase parser is instantiated, language is provided as a parameter which then applies the language-specific lexical redundancy rules to all lexical elements that make up the speaker model. The main script creates a separate speaker model for speakers of any language present in the lexicon. These speaker models differ only in terms of the composition of (some) lexical items; the computational core remains the same. We imagine the brain model as a container that hosts all modules and their connections, as shown in Figure 32.

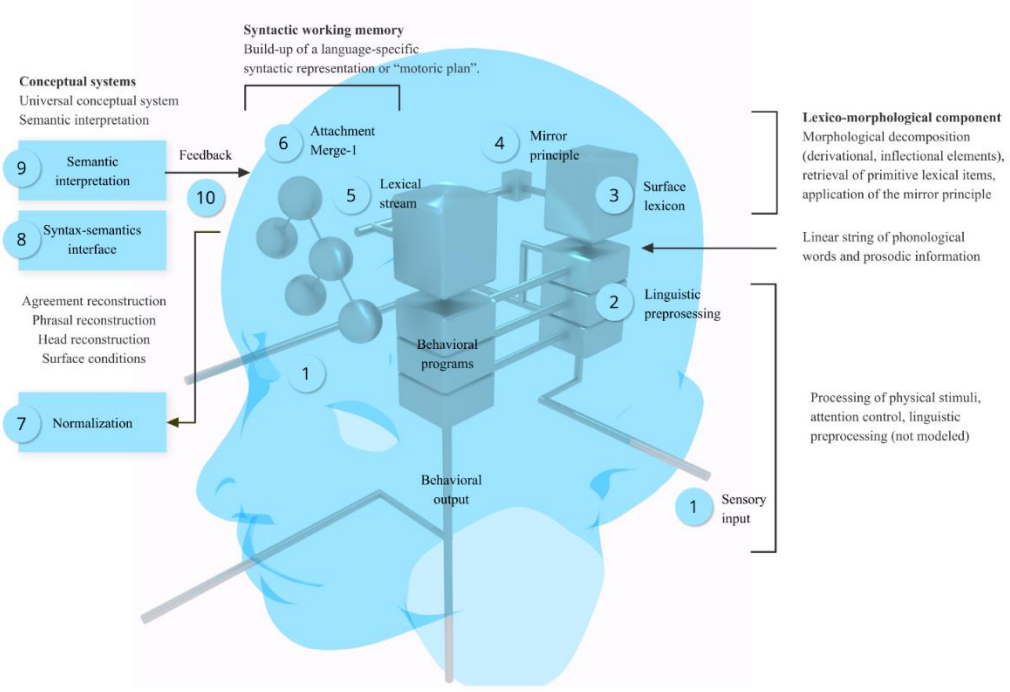


Figure 32. The speaker model contains all submodules defined by the theory and their connections.

8.1.2 Parse sentence (parse_sentence)

When the main script wants to parse a sentence, it sends the sentence to the speaker model (linear phase parser) as a list of words. The parser function prepares the parser by setting a host of input parameters (mostly having to do with logging and other support functions), and then calls for the recursive parser function *prase_new_item* with four arguments *current structure*, *list*, *index* and *inflection_buffer*, with *current structure* being empty, *index* = 0 and *inflection_buffer* empty.

```
85      def parse_sentence(self, count, lst):
86          self.sentence = lst
87          self.start_time = process_time()
88          self.initialize()
89          self.plausibility_metrics.initialize()
90          self.narrow_semantics.initialize()
91          log_new_sentence(self, count, lst)
92          self.parse_new_item(None, lst, 0)
```

This function is self-explanatory. The last function (line 92) starts the recursive parsing operation.

8.1.3 Recursive parsing function (parse_new_item)

The recursive parsing function takes the currently constructed phrase structure α, a linearly ordered list of words, an index in the list of words and an inflection buffer as its arguments. These objects therefore represent the contents of the syntactic working memory at any processing phase.

It will first check if there is any reason to terminate processing. Processing is terminated if there are no more words, or if a self-termination flag *self.exit* has been raised somewhere during the execution.

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

```
133     def circuit_breaker(self, ps, lst, index):
134         set_logging(True)
135         if self.exit:
136             return True
137         if index == len(lst):
138             self.complete_processing(ps)
139             return True
140         self.time_from_stimulus_onset = int(len(lst[index]) * 10)
141         if not self.first_solution_found:
142             self.resources['Total Time']['n'] += self.time_from_stimulus_onset
```

If there are no more words, contents of the syntactic working memory are send out for interpretation. This function is *complete_processing*. Suppose, however, that a new word w was consumed. At this point each word is phonological string. To retrieve properties of w, the lexicon will be accessed.

```
97             retrieved_lexical_items = 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 and will be explored in some order. The ordered list will be added to the recursive loop as an additional layer.

```
98             for lexical_constituent in retrieved_lexical_items:
```

The lexicon is a mapping from phonological surface stings (keys) into constituents. If the lexical entry is mapped into a morphological decomposition, the resulting constituent will “contain” the decomposition and no other properties. It constitutes a morphological chunk that will not and cannot enter syntax in this form; we can imagine these chunks are containing pointers to other elements in the lexicon. If the lexical entry maps into a primitive lexical item, then the constituent, primitive lexical item, will contain the set of features as specified in its lexical entry. If the lexical entry maps into an inflectional morpheme, then it will again consist of a set of (inflectional) features, but these will be processed differently. The following figure illustrates the idea.

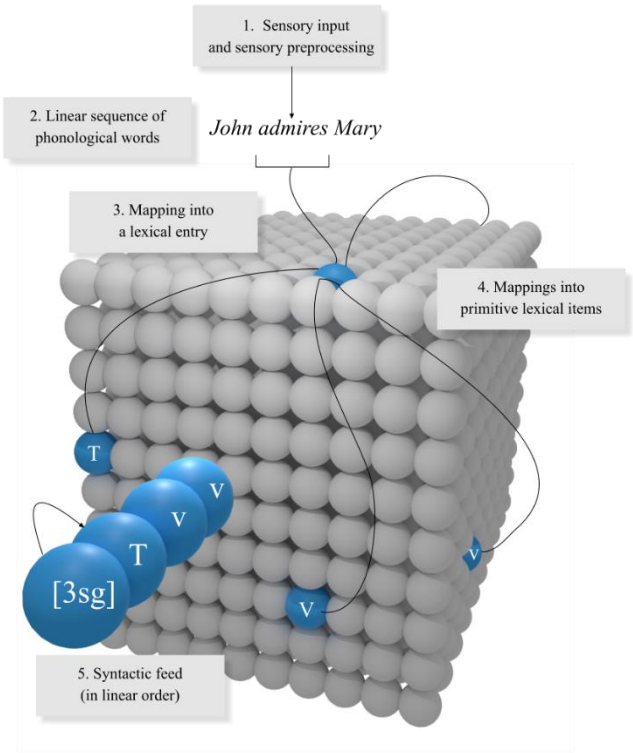


Figure 40. Organization of the lexicon. The lexicon is a mapping from phonological surface strings into constituents which make up the lexicon. Some surface strings map into primitive lexical items (blue items) which are sets of lexical features (e.g., /the/ ~ D). Other surface strings map into morphological chunks which contain pointers to further items (morphological decompositions)(e.g., /admires/ ~ V, v, T, 3sg).

We use the phrase structure class to generate lexical entries, whether they are morphologically complex or not. The intuitive motivation for this assumption is that at the bottom level the lexicon has to map something into primitive lexical items; morphological chunks are just an additional layer build on the top of that foundation.

Next an item from this list of possible lexical entries will be subjected to *morphological parsing*. Notice that a lexical item returned by the lexical retrieval may still consists of a morphological decomposition. The morphological parser will change the list of words that now contains the individual morphemes that were part of the original input word, in reverse order, together with the lexical item corresponding with the first item in the new list.

```
98          for lexical_constituent in retrieved_lexical_items:
99              self.morphology.morphological_parse(ps, lexical_constituent, lst.copy(), index, inflection_buffer)
```

All word-internal morphemes are used to modify the original list of words, which now contains the morphological decomposition of the word in addition to the original phonological words. An input list *John* + *admires* + *Mary* will be transformed into *John* + *3sg* + *T* + *v* + *admire* + *Mary* (ignoring the processing of the proper names). Finally, the *first item* in the new list will be sent to the syntactic component via lexical stream.

```
100                 self.lexical_stream.stream_into_syntax(lexical_constituent, lst, ps, index, inflection_buffer)
```

The lexical stream pipeline handles several operations. Some of the morphemes could be inflectional, in which case they are stored as features into a separate inflectional 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. To allow backtracking from inflectional processing, the buffer is forwarded to the recursive parsing function as an input parameter. 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. Other lexical items enter the syntactic module.

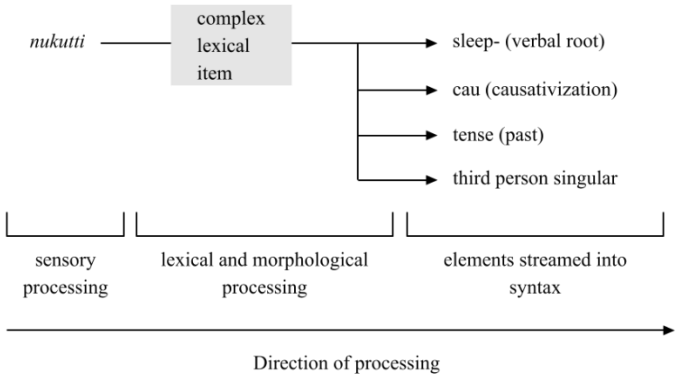


Figure 41. Lexical processing pipeline.

They 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 ranked.

```
101                merge_sites = self.plausibility_metrics.filter_and_rank(ps, lexical_constituent)
```

Each site from the ranking is then explored.

```
102            for site, transfer, address_label in merge_sites:
103                log(f'\n\t\t{address_label}')
104                new_constituent = self.attach(ps.target_left_branch(site), site, lexical_constituent, transfer)
105                self.working_memory.remove_items(merge_sites)
106                self.parse_new_item(new_constituent.top(), lst, index + 1)
107                if self.exit:
108                    break
```

The loop examines each (site, transfer) pair provided by the plausibility function above. The candidate sites are right edge nodes in the partial phrase structure currently being developed. Transfer is a Boolean variable determining whether we want to transfer the left branch or not before Merge. The operation targets a node and attaches the incoming lexical item to it. The result will then be passed recursively to the parsing function.

The code above contains two essential functions: *target_left_branch* and *attach*. Recursive branching requires that we create a new structure when some solution is considered, which presupposes that we can identify equivalent nodes between these structures. The function *attach* performs Merge-1 and sinking depending on the situation and performs working memory operations.

```
113        def attach(self, left_branch, site, terminal_lexical_item, transfer):
114            self.working_memory.maintain(site)
115            if left_branch.belong_to_same_word(site):
116                new_constituent = left_branch.sink_into_complex_head(terminal_lexical_item)
117            else:
118                new_constituent = self.attach_into_phrase(left_branch, terminal_lexical_item, transfer)
119            self.consume_resources("Merge", terminal_lexical_item)
120            log('\n')
121            return new_constituent
```

Once all words have been processed, the result will be submitted to a finalization stage implemented by function *complete_processing*. This process will apply transfer, LF-legibility and semantic interpretation.

```
144     def complete_processing(self, ps):
145         self.transfer.transfer_to_LF(ps)
146         if self.postsyntactic_tests(ps):
147             self.resources.update(PhraseStructure.resources)
148             report_success(self, ps)
149         else:
150             self.narrow_semantics.reset_for_new_interpretation()
151             report_failure(ps)
152         if not self.first_solution_found:
153             self.consume_resources("Garden Paths", ps)
```

If these fail, the result is rejected; otherwise processing continues. Once a solution has been accepted and documented, control is returned to the parsing recursion.

8.2 Psycholinguistic plausibility

8.2.1 General architecture

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

```
101                     merge_sites = self.plausibility_metrics.filter_and_rank(ps, lexical_constituent)
```

The parser will get a filtered and ranked list in return, which is used by the parser to explore solutions.

```
102                 for site, transfer, address_label in merge_sites:
103                     log(f'\n\t\t{address_label}')
104                     new_constituent = self.attach(ps.target_left_branch(site), site, lexical_constituent, transfer)
105                     self.working_memory.remove_items(merge_sites)
106                     self.parse_new_item(new_constituent.top(), lst, index + 1)
107                     if self.exit:
108                         break
```

The filter_and_rank function contains the following core iteration.

```
79      elif ps.word_internal() and self.dispersion_filter_active():
80          solutions = [(ps.bottom(), True, self.generate_address_label())]
81      else:
82          self.brain_model.working_memory.in_active_working_memory, self.brain_model.working_memory.not_in_active_working_memory =
83              self.brain_model.working_memory.active_working_memory_catalog(ps)
84          log(f'\n\t\tFiltering and ranking merge sites...')
85          nodes_available = self.filter(self.brain_model.working_memory.in_active_working_memory, w)
86          merge_sites = self.rank_merge_right_(nodes_available, w)
87          all_merge_sites = merge_sites + self.brain_model.working_memory.not_in_active_working_memory
88          solutions = self.evaluate_transfer(all_merge_sites)
```

If the incoming word was part of the previous word, it will always be merged as its sister (79–80). We will therefore only return one solution. Otherwise, filter (line 85) and ranking (line 86) will be applied, in that order. Only nodes in the active working memory (line 82–83) are processed by using filter and ranking; the rest are added to the solutions list in random order (line 87). Only nodes that passes the filter are ranked. Ranking uses cognitive parsing heuristics, as explained below. The user can activate and inactivate these heuristics in the configuration file.

8.2.2 Filtering

Filtering is implemented by going through all available nodes (i.e. those which are in the active working memory) and by rejecting them if a condition is satisfied. When a parsing branch is closed by filtering, it can never be explored by backtracking. 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.

```
252     def filter(self, list_of_sites_in_active_working_memory, w):
253         adjunction_sites = []
254         for N in list_of_sites_in_active_working_memory:
255             if N.does_not_accept_any_complements():
256                 log(f'Reject {N} + {w} because {N} does not accept complementizers. ')
257                 continue
258             if N.complex() and self.left_branch_filter(N):
259                 log(f'Reject {N} + {w} due to bad left branch ({self.brain_model.LF.error_report_for_external_callers})...')
260                 continue
261             if self.word_breaking_filter(N, w):
262                 log(f'Reject {N} + {w} because it breaks words. ')
263                 continue
264             if N.impossible_sequence(w):
265                  log(f'Reject {N} + {w} because the sequence is impossible. ')
266                  continue
267             adjunction_sites.append(N)
268         return adjunction_sites
```

The left branch filter sends the phrase structure through the LF-interface and examines if transfer is successful. If it is not successful, the parsing path will be closed.

```
270     def left_branch_filter(self, N):
271         set_logging(False)
272         dropped = self.brain_model.transfer.transfer_to_LF(N.copy())
273         left_branch_passes_LF = self.brain_model.LF.LF_legibility_test(dropped, self.left_branch_filter_test_battery)
274         if not left_branch_passes_LF:
275             log(f'in {dropped}. ')
276         set_logging(True)
277         return not left_branch_passes_LF
```

8.2.3 Ranking (rank_merge_right_)

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

```
209         # Create baseline default weighting order (default order is decided by input parameters in study config file)
210         self.weighted_site_list = self.create_baseline_weighting([(site, 0) for site in site_list])
211         calculated_weighted_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 examined and evaluated in the light of plausibility conditions which, when they apply, increase or decrease the weight provided for each site in the list. The function returns a list where 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 condition, 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:

```
214         for site, weight in self.weighted_site_list:
215             new_weight = weight
216             for key in self.plausibility_conditions:
217                 if self.plausibility_conditions[key]['condition'](site):
218                     log(self.plausibility_conditions[key]['log'] + f' for {site}...')
219                     log('('+str(self.plausibility_conditions[key]['weight'])+') ')
220                     new_weight = new_weight + self.plausibility_conditions[key]['weight']
221             calculated_weighted_site_list.append((site, new_weight))
```

In the current version the weight modifiers are ±100. These numbers outperform the small numbers assigned by the baseline weighting. 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 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.

8.2.4 Knocking out heuristic principles

Heuristic principles can be knocked and/or controlled by *config_study.txt*. This is useful feature when we attempt to develop the principles, making it possible to observe their effects in isolation or in combination with other principles.

```
self.plausibility_conditions = \
    {'positive_spec_selection':         {'condition': self.positive_spec_selection,
                                         'weight': self.brain_model.local_file_system.settings.get('positive_spec_selection', 100),
                                         'log': '+Spec selection'},
     'negative_spec_selection':         {'condition': self.negative_spec_selection,
                                         'weight': self.brain_model.local_file_system.settings.get('negative_spec_selection', -100),
                                         'log': '-Spec selection'},
     'break_head_comp_relations':       {'condition': self.break_head_comp_relations,
                                         'weight': self.brain_model.local_file_system.settings.get('break_head_comp_relations', -100),
                                         'log': 'Head-complement word breaking condition'},
     'positive_head_comp_selection':    {'condition': self.positive_head_comp_selection,
                                         'weight': self.brain_model.local_file_system.settings.get('positive_head_comp_selection', 100),
                                         'log': '+Comp selection'},
     'negative_head_comp_selection':    {'condition': self.negative_head_comp_selection,
                                         'weight': self.brain_model.local_file_system.settings.get('negative_head_comp_selection', -100),
                                         'log': '-Comp selection'},
     'negative_semantics_match':        {'condition': self.negative_semantic_match,
                                         'weight': self.brain_model.local_file_system.settings.get('negative_semantics_match', -100),
                                         'log': 'Semantic mismatch'},
     'lf_legibility_condition':         {'condition': self.lf_legibility_condition,
                                         'weight': self.brain_model.local_file_system.settings.get('lf_legibility_condition', -100),
                                         'log': '-LF-legibility for left branch'},
     'negative_adverbial_test':         {'condition': self.negative_adverbial_test,
                                         'weight': self.brain_model.local_file_system.settings.get('negative_adverbial_test', -100),
                                         'log': '-Adverbial condition'},
     'positive_adverbial_test':         {'condition': self.positive_adverbial_test,
                                         'weight': self.brain_model.local_file_system.settings.get('positive_adverbial_test', 100),
                                         'log': '+Adverbial condition'}
     }
```

8.3 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 paths. 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 during the first pass parse without 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. Grammatical operations are counted as "black boxes" in the sense that we ignore all internal operations (e.g., minimal search, application of merge, generation of rejected solutions). The number of head reconstructions, for example, is increased by one if and only if a head is moved from a starting position X into a final position Y; all intermediate positions and rejected solutions are ignored. This therefore quantifies the number of "standard" head reconstruction operations – how many times a head was reconstructed – that have been implemented during the processing of an input sentence. The number of all computational steps required to implement the said black box operation is always some linear function of that metric and is ignored. For example, countercyclic merge operations executed during head reconstruction will not show up in the number of merge operations; they are counted as being "inside" one successful head reconstruction operation. It is important to keep in mind, though, that each transfer operation will potentially increase the number independently of whether the solution was accepted or rejected. For example, when the left branch α is evaluated during [α β], the operations are counted irrespective of whether α is rejected or accepted during the operation.

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

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

```
71      self.resources = {"Total Time": {'ms': 0, 'n': 0},     # Count pre
72                        "Garden Paths": {'ms': 0, 'n': 0},
73                        "Merge": {'ms': 5, 'n': 0},
74                        "Head Chain": {'ms': 5, 'n': 0},
75                        "Phrasal Chain": {'ms': 5, 'n': 0},
76                        "Feature Chain": {'ms': 5, 'n': 0},
77                        "Agree": {'ms': 5, 'n': 0},
78                        "Feature": {'ms': 5, 'n': 0},
79                        "Scrambling Chain": {'ms': 5, 'n': 0},
80                        "Extraposition": {'ms': 6, 'n': 0},
81                        "Last Resort Extraposition": {'ms': 5, 'n': 0},
82                        "Mean time per word": {'ms': 0, 'n': 0}
83                        }
```

If the researcher adds more entries to this dictionary, they will 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. 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.

References

Alexiadou, A., & Anagnostopoulou, E. (1998). Parametrizing AGR: Word Order, V-movement and EPP Checking. *Natural Language and Linguistic Theory.*, *16*(3), 491–539. https://doi.org/10.1023/a:1006090432389

Bianchi, V., & Chesi, C. (2010). Reversing the perspective on Quantifier Raising. *Rivista Di Grammatica Generativa*, *35*, 3–38.

Bianchi, V., & Chesi, C. (2014). Subject islands, reconstruction, and the flow of the computation. *Linguistic Inquiry*, *45*(4), 525–569.

Brattico, P. (2020). Finnish word order: does comprehension matter? *Nordic Journal of Linguistics*, *44*(1), 38–70. https://doi.org/doi:10.1017/S0332586520000098

Brattico, P. (2021a). A dual pathway analysis of information structure. *Lingua*, *103156*.

Brattico, P. (2021b). Null arguments and the inverse problem. *Glossa: A Journal of General Linguistics*, *6*(1), 1–29. https://doi.org/https://doi.org/10.5334/gjgl.1189

Brattico, P. (2022a). Predicate clefting and long head movement in Finnish. *Linguistic Inquiry*, *54*(4), 663–692. https://doi.org/http://dx.doi.org/10.1162/ling_a_00431

Brattico, P. (2022b). Structural case assignment, thematic roles and information structure. *Studia Linguistica*, *76*(3), XX–XX. https://doi.org/10.1111/stul.12206

Brattico, P. (2023a). Across the board agreement in Finnish. *Manuscript Submitted for Publication*.

Brattico, P. (2023b). Computational analysis of Finnish nonfinite clauses. *Nordic Journal of Linguistics*, *X*(X), X–X.

Brattico, P. (2023c). EPP and Agree at the edge. *Ms*.

Brattico, P., & Chesi, C. (2020). A top-down, parser-friendly approach to operator movement and pied-piping. *Lingua*, *233*, 102760. https://doi.org/10.1016/j.lingua.2019.102760

Cann, R., Kempson, R., & Marten, L. (2005a). *The Dynamics of Language: An Introduction (Syntax and Semantics, Volume 35)*. Elsevier Academic Press.

Cann, R., Kempson, R., & Marten, L. (2005b). *The Dynamics of Language: An Introduction (Syntax and Semantics, Volume 35)*. Elsevier Academic Press.

Chesi, C. (2004). *Phases and cartography in linguistic computation: Toward a cognitively motivated computational model of linguistic competence*.

Chesi, C. (2012). *Competence and Computation: toward a processing friendly minimalist Grammar*. Unipress.

Chesi, C. (2013). Do the "right" move. *Studies in Linguistics*, *6*, 107–138.

Chomsky, N. (1986). *Knowledge of Language: Its Nature, Origins and Use*. Praeger.

Chomsky, N. (2000). Minimalist Inquiries: The Framework. In R. Martin, D. Michaels, & J. Uriagereka (Eds.), *Step by Step: Essays on Minimalist Syntax in Honor of Howard Lasnik* (pp. 89–156). MIT Press.

Chomsky, N. (2001). Derivation by Phase. In M. Kenstowicz (Ed.), *Ken Hale: A Life in Language* (pp. 1–37). MIT Press.

Chomsky, N. (2004). Beyond explanatory adequacy. In A. Belletti (Ed.), *Structures and Beyond: The Cartography of Syntactic Structures, volume 3* (pp. 104–131). Oxford University Press.

Chomsky, N. (2008). On Phases. In C. Otero, R. Freidin, & M.-L. Zubizarreta (Eds.), *Foundational Issues in Linguistic Theory: Essays in Honor of Jean-Roger Vergnaud* (pp. 133–166). MIT Press.

Heycock, C. (1991). *Layers of Predication: the Non-Lexical Syntax of Clauses* [PhD dissertation]. University of Pennsylvania.

Kayne, R. (1983). Connectedness. *Linguistic Inquiry, 14.2*, 223–249.

Kayne, R. (1984). *Connectedness and Binary Branching*. De Gruyter Mouton.

Kempson, R., Meyer-Viol, W., & Gabbay, D. M. (2001). *Dynamic syntax: The flow of language understanding*. Wiley-Blackwell.

Kiss, K. É. (2002). The EPP in a topic-prominent language. In P. Svenonius (Ed.), *Subjects, Topics, and the EPP*. Oxford University Press.

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. MIT Press.

Phillips, C. (1996). Order and structure [Ph.D. thesis]. In *MIT*.

Rosengren, I. (2002). EPP: A syntactic device in the service of semantics. *Studia Linguistica, 56*, 145–190.

Rothstein, S. (1983). *The syntactic forms of predication*.

Williams, Edwin. (1980). Predication. *Linguistic Inquiry., 11*, 203–238.

Zwart, J.-W. (2009). Prospects for top-down derivatio. *Catalan Journal of Linguistics, 8*, 161–187.

Zwart, J.-W. (2015). Top-down derivation, recursion, and the model of grammar. *Syntactic Complexity across Interfaces, 3*, 25–42.