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

(version 20.1)

2019

(Revised June 2025)

Pauli Brattico

Abstract

This document describes a computational implementation of linear grammar phase (LPG). The model assumes that the core computational operations of narrow syntax are applied incrementally on a phase-by-phase basis in language comprehension and production, with transformational operations implemented both cyclically and noncyclically. 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 20.1). IUSS, Pavia.

1 Introduction

This document describes a computational Python-based implementation of a linear phase grammar (LPG) 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 and production. It captures cognitive principles of language ("competence") and cognitive mechanisms involved in language comprehension and use ("performance").

This document describes properties of the version 20.1, which keeps within the framework of the original work but provides major 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.[2]

---

[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).

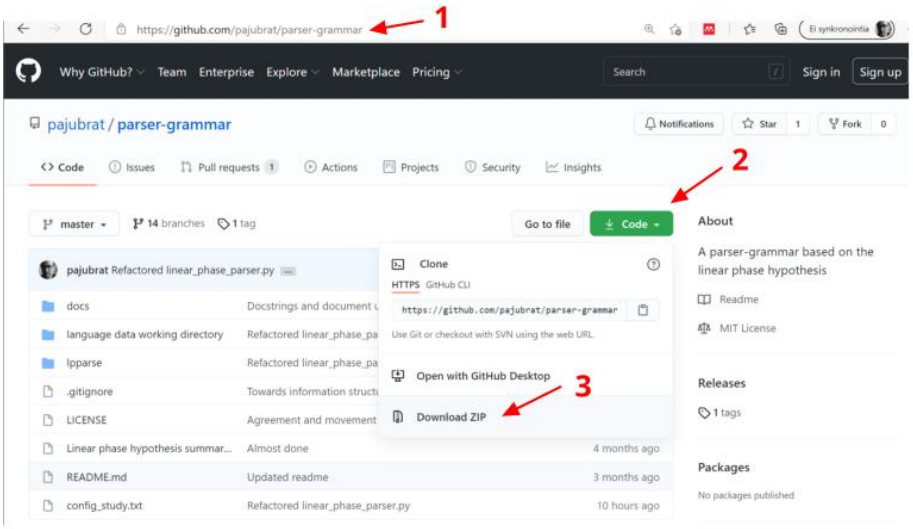
[2] A brief history of this work is as follows. Scientific justification in the advanced natural sciences depends on the notion of rigorous proof, in which the data is calculated (deduced) from the hypothesis or theory (Brattico, 2021b, 2021c, 2024). The purpose of the software described here is to constructs proofs of this type for certain range of linguistic hypotheses. Originally the computational infrastructure was set up in order to verify what happens when a certain left-to-right model of language competence and performance was applied to real linguistic data (Brattico, 2020, 2021b, 2021a, 2021d, 2023b, 2023c; Brattico & Chesi, 2020). Later it became

clear that the original focus on the particular grammatical model was sometimes irrelevant, though not completely irrelevant: most of the operations defined in the model (though not all) could be reverse-engineered without loss of empirical or theoretical context within the more traditional bottom-up models. This step led to a considerable abstraction in the model and its use in concrete linguistic work, and consequently the particular left-to-right model ceased to be the main focus of the enterprise (Brattico, 2022, 2024, 2025). Separate entry-level scripts were written for traditional bottom-up models (Brattico, 2024). I have explained the methodology and how to set-up and apply it in a series of video presentations, with the concrete implementation discussed in the context of Python as is the case here. See `https://www.youtube.com/channel/UCa0vhL8xC5aSH8uvZgF6PgA`.

2 Installation and use

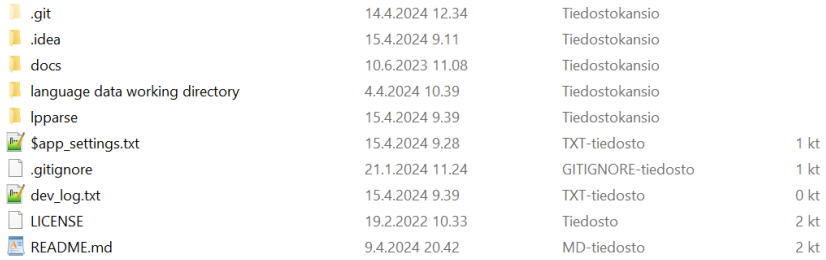
2.1 Installation

The linear phase grammar algorithm is a collection of Python functions that processes (comprehends, produces) 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 a separate directory. The same files and folders displayed on the above figure should appear in that directory. Because the script is written in Python (3x) programming language, the user must have Python installed on the local

machine.[3] 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 particular studies, and `/lpparse` stores the modules containing the program code. `$app_settings.txt` is a text file that configures the script, in a manner explained below. In most cases the script comes preconfigured so that it runs some particular study and can therefore be tested without providing further information or parametrization.

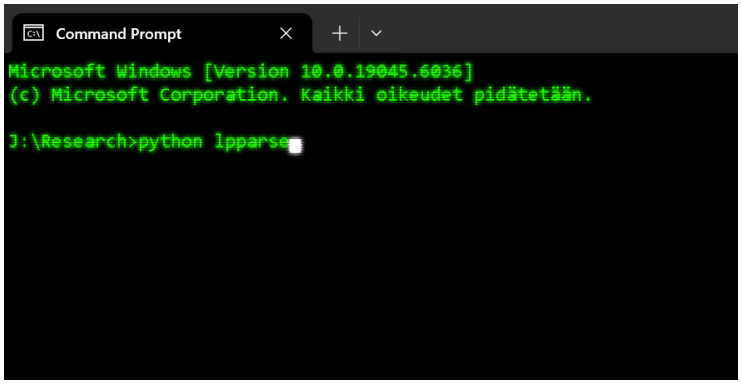
2.2 Use

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.[4] 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`. If the local installation directory is `J:\Research` (as it is on the author's local machine in the example below), the script is executed as follows:

---

[3] 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."

[4] 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.



The program reads a study configuration file `$app_settings.txt` from the local installation directory which points to 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. For example:

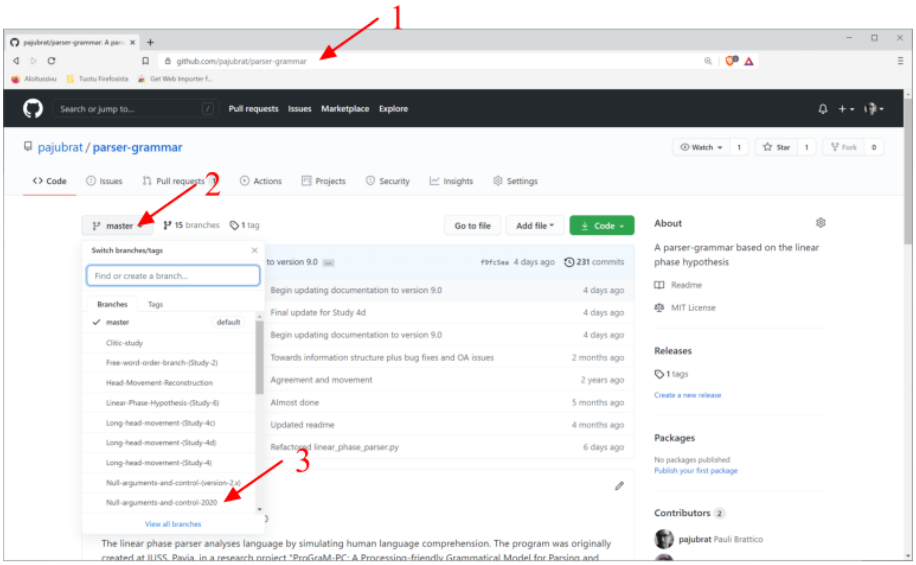
```
& 1. Initial binding condition tests (English)

    # Condition C (1-3)

        John admires Bill
        ¦-> Binding: John[a] admire Bill[b]

        he `s brother admires Bill
        ¦-> Binding: he[b] brother[a] admire Bill[b,c]

        Bill said that John admires Tim
        ¦-> Binding: Bill[a] say John[b] admire Tim[c]

    # Condition A (4-6)

        John admires himself
        ¦-> Binding: John[a] admire self[a]

        he `s brother admires himself
        ¦-> Binding: he[b] brother[a] admire self[a]

        Bill said that John admires himself
        ¦-> Binding: Bill[a] say John[b] admire self[b]
```

This is a screenshot of the dataset (a list of sentences with comments marked with #) the script processes (contents of the dataset file depend on the study). The results are generated into several *output files* inside the study folder. These output files record the results of the execution of the script and include such information as how many mistakes the script did, how it derived various sentences, what type of semantic interpretations it associated with each sentence, and so on. An

alternative way to use the program is to launch the graphical user interface by command `python lpparse -app`. This is described in Chapter 9.

2.3 Replication

When a study is published, the input files plus its source code is stored in the 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 be branched, meaning that it is possible to develop several versions of the software in parallel (and possibly 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 to the source code repository (1, Figure below), click for the tab shown in the figure below (2) and select the branch that is of interest (3).



This should select a snapshot of a development branch that contains a version of the script that was used in a published study. Use of these branches is not recommended for anything else than replication.

The baseline model that is being developed at any given time is always aimed at modeling some dataset, usually from Finnish. The software is not a general purpose tool, and not something other researchers should branch and develop. It incorporates idiosyncratic and author-specific assumptions about language and language processing. A better approach to develop these methods and modelling is to write one's own computational infrastructure from scratch, as I have described in the YouTube videos.[5]

[5] https://www.youtube.com/channel/UCa0vhL8xC5aSH8uvZgF6PgA.

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 to 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] Similarly, language production is possible "out of the blue" without external stimulus or prompt, and is in principle free from external control, a point Chomsky (1959) famously made in connection with his review of Skinner's *Verbal Behavior*.

Let us consider language comprehension first. 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, perhaps best captured by making a simple distinction between attested and unattested linguistic behavior. 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*. More broadly, a scientific theory must be

[6] This does not mean that language processing does not depend on context. Some aspects of semantic interpretation, such as the intended denotations of pronouns, depend on context; but if no context is present, the sentence receives an all-new reading.

observationally adequate, and the fact that it is observationally adequate must be shown by deductive calculation. Because the model judges input sentences, it satisfies observational adequacy by replicating native speaker comprehension and thus it implements a *recognition grammar*.[7]

Some aspects of the comprehension model are language-specific, others are universal and depend on the biological structure of the human brain. A universal property can be elicited from speakers of any language. For example, it is a universal property that an interrogative clause cannot be formed by moving an interrogative pronoun out of a relative clause (as in, e.g., **who did John met the person that Mary admires__?*). On the other hand, it is a property of Finnish and not English that singular marked numerals other than 'one' assign the partitive case to the noun they select (*kolme sukka-a* 'three.sg.0 sock.sg.par'). The latter are acquired from the input during language acquisition. Universal properties plus the storage systems for language-specific knowledge constitute the fixed portion of the grammar, 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

---

[7] See https://youtu.be/2JePzDkXAT4

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 (i) fixed, universal components and (ii) 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 learning, exposure and other experiences. An explanatory adequate grammar will posit just enough fixed universal structure and individual variation to capture the properties of all languages. This could be accompanied by a formal learning theory which creates speaker models from linguistic (and other) experiences.[8]

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

[8] See `https://youtu.be/WFdElB99aZk` and `https://youtu.be/04SnQLVugLA`

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 and organized 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 meaning, let alone meaning in the broadest sense. 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 semantic predictions. We can leave the 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.[9]

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

---

[9] 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.

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 indeed many imaginable alternatives, is to examine to what extent they will generate correct semantic interpretations for a set of input sentences in a dataset or several; it is pointless to try to use any other justification or argument either in favor or against any specific model or assumption. In general, then, we do not posit further restrictions or properties to the syntax-semantic interface apart from its existence.[10]

The LPG model described in this document also applies to language production. The idea is that language production proceeds from left-to-right and word-by-word basis, cyclically in tandem with the actual production, but instead of generating ambiguous surface level expressions the generative engine targets a level that is usually referred as the PF-interface, which then generates both actual language production (speech, typing, writing) and its meaning, the latter which is matched with the "intended" message. So the processing branches off from the PF-interface.

The left-to-right cycle of LPG differs from the more standard bottom-up model in which grammatical expressions are derived by combining lexical items and the resulting complex syntactic objects recursively into larger units. The direction of the derivation is therefore not from

---

[10] See `https://youtu.be/5e8mzIGP-Zw` for discussion of how to work with semantics in connection with unambiguous datasets.

left to right (in relation to the surface expression) but from bottom to up. The two models have considerable overlap since they can generate the same phrase structure representations and because the left-to-right system includes several bottom-up operations in its arsenal, such as a bare Merge which combines X and Y into [$_{XP}$ X Y]. Several core computational operations proposed in the LPG (such as internal and external search) apply to phrase structures generated by a bottom-up model simply in virtue of the fact that they apply to phrase structure representations which both systems generate. Both models can described the same abstract mappings between phrase structure representations. This leads me to believe that we should understand the left-to-right model as an extension of the traditional bottom-up cycle or, in other words, the bottom-up model as an idealization or abstraction of the real system.

3.2 Computational generative grammar

Scientific hypotheses must be justified by deducing the observed facts from the proposed hypothesis. The data, deductive calculations, and the theories and hypotheses must be provided in an unambiguous form (1).

(1) *Methodological principle*

Data, theories and the deductive chains connecting the two must be unambiguous.

We say that a grammar or grammatical theory that satisfies (1) is a *computational generative grammar* (CGG). The linear phase grammar (LPG) discussed in detail in this document is one computational generative grammar among many other possible computational generative grammars.

Let us consider how to eliminate ambiguity from linguistics. A linguistic theory or grammar is a system that defines a set of attested (grammatical) expressions in some human language and only them (Section 3.1). In addition to defining grammatical expressions, we expect that the grammar provides grammatical expressions with additional properties such as syntactic analyses, semantic interpretations, felicitous pragmatic contexts and other properties, depending on the scope of each

individual study. Let us say that the surface expressions (suppose they are linear sequences of phonological words) and their further syntactic, semantic and pragmatic properties are jointly called *expressions*. One way to construct a grammatical theory satisfying (1) is to provide a set of grammatical rules together with lexical items, which generate grammatical expressions by applying the rules to the lexical items and their own output. We call a grammatical theory of this type an *enumerative grammar*. An enumerative grammar defines the set of grammatical expressions by literally generating it. The computational entry scripts that can do this are provided in (Brattico, 2024, 2025).

Another possibility – implemented by the linear phase grammar described here – is to formulate the grammatical theory in the form of a "filter" which takes expressions as input and recognizes whether they are grammatical or not and, if they are, provides them with the further attributes we are interested in any particular study (semantic interpretation etc.). This type of theories can be called *recognition grammars*. Despite the different ways these grammars work, both enumerative and recognition grammars are computational generative grammars: they define sets of attested expressions and provide them with further properties. An enumerative grammar defines the set by enumerating its positive members, while the recognition grammar defies it by implementing its characteristic function.[11]

Suppose, then, that we have constructed a computational generative grammar G that is either enumerative grammar or recognition grammar, and that we have some expression E in our dataset, E being either grammatical or ungrammatical. How can we show that G entails E? Suppose that G is formulated as an enumerative grammar. If E is grammatical, we must demonstrate that there is a chain of deductive calculations, call it *derivation*, which begins with the axioms of the theory and the relevant lexical items and ends up with E. We can accomplish this by exploring recursively all possible derivations (from the relevant lexical set) until E is encountered. A

[11] See `https://youtu.be/WwNDBut0ZE0` and the following presentations for introduction and further explanation.

function that performs this recursive operation is called a *derivational search function*. If E is ungrammatical, we must establish that there is no derivation that entails E. A computational generative grammar is a system consisting of the lexical items, grammar and the executive functions which implements the above checking procedure. The *dataset* will contain all expressions E. . . that we think bear on the hypothesis we want to test. The general idea can be illustrated as follows:

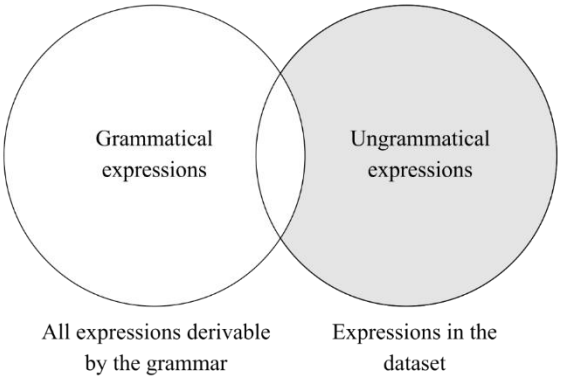


Figure 1. The computational generative grammar framework: the grammar defines the set of attested (and only attested) expressions in the dataset.

The data are now unambiguous, since we have provided them in the form of an explicit dataset; the theory or hypothesis is unambiguous, since it is formalized, here provided in a machine-readable form; and the justification of the theory or hypothesis by the data is unambiguous, since every expression in the dataset is either linked with the theory/hypothesis by means of a completely explicit derivation or by an exhaustive search demonstrating that no derivation exists.

Recognition grammar implements a characteristic function for the set of attested expressions. We can construct a trivial recognition grammar by using an enumerative grammar as a platform: dissolve each input expression E into constituent lexical items and examine whether E can or cannot be derived. Successful derivations generate syntactic analyses and semantic interpretations for E. But because the recognition grammar implements a characteristic function for the set of grammatical expressions, it does not have to enumerate all expressions that can be derived from

the same words; it can recognize (and reconstruct) the properties of E by inspecting E alone. The system is therefore the same as the one provided in Figure 1, except that "all expressions derivable by the grammar" is replaced with "all expressions accepted by the grammar."

There is a subtle difference between the enumerative and recognition grammars when it comes to practical linguistic work. An enumerative grammar generates a set of grammatical expressions from lexical items, and is evaluated against data by making sure that unattested expressions are not generated. Anything not generated by the grammar is assumed to be ungrammatical. A recognition grammar must be tested explicitly against all relevant expressions, including ungrammatical ones, and the force of the justification depends on the number and type of expressions in the dataset.[12]

[12] It is possible to use bare lexical selection in the testing of a recognition grammar: we test the algorithm against all linear permutations of the input words. This functionality is included in the linear phase model (LPG), although it has not been used in any published study to date.

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 LPG 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 mostly in the published literature, though some comments have been added when deemed useful. The level of detail provided is sufficient only for broader understanding of the basic logic of the theory and its implementation.

The purpose of the system is not to perform parsing in the engineering sense or to create a linguistic model out of the blue; rather, its purpose is to describe and explain linguistic data. The main motivation for its properties is always a linguistic dataset that reflects some phenomenon of interest rather than human imagination or some engineering consideration. Thus, anybody developing a system of this kind must have a dataset available at all times for testing purposes so that all design decisions can be tested and verified by running them against empirical data. That a testing procedure of this kind exists when developing a computational generative grammar, whether along the lines proposed here or by following a radically different blueprint, will be assumed throughout.[13]

[13] At the early stages of the development the dataset does not need to be large, systematic or gapless; it is sufficient to test the proposed model by relying on a few representative examples. In a real linguistic project, however, the dataset must be complete in some well-defined sense, putting considerable requirements on the model.

4.2 Merge-1

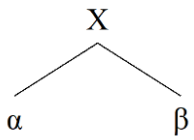
Linguistic input is received by the hearer in the form of sensory stimulus. We can first think of the input, to simplify the situation to the bare essentials, as a one-dimensional string $\alpha \oplus \beta \oplus \ldots \oplus \gamma$ of phonological words. In order to understand what the sentence 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 $\alpha \oplus \beta$, Merge yields $[\alpha, \beta]$(2).

(2) 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 $[\alpha, \beta]$, α is the left constituent, β the right constituent.

What do we mean by saying that the constituents are "put together"? 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 "one" item, a 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. This presupposes that there exists a formally

defined notion of phrase structure that is able to represent entities like (2). How it is implemented is nontrivial from the linguistic point of view, but trivial as an engineering task.[14]

Example (2) suggests that the syntactic component combines phonological words. While this is a possibility, it is not linguistically useful. We assume that (2) 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* such as lexical categories (noun, verb), inflectional features ('third person singular') and meaning ('John', 'sleeping')(§ 4.9.5). We assume from this point on that syntax operates with lexical items, not with phonological words.[15] To keep things simple, phonological words will substitute for the lexical items in the illustrations in this document.

The resulting complex chunk [α, β] is asymmetric and has a *left (or first) constituent* and a *right (second) constituent*. The terms "left"/"first" and "right"/"second" 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 when reading linguistic expressions, (2) implies that the constituent that arrives first will be the left constituent.

The theory itself places no limit on the number of elements that can be assembled together, although there are several other cognitive bottlenecks that can limit the resulting structures and our ability to process them. Suppose the next word is *furiously*. Example (3) shows three possible attachment sites, all which correspond to different hierarchical relations between the words.

---

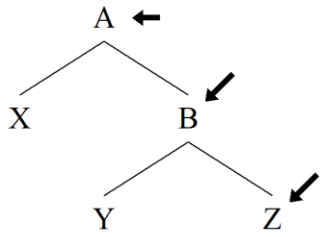
[14] Python-implementation is discussed in `https://youtu.be/UQedH-mJ5UE`, which presents one possible starting point.

[15] It is possible to omit the phonology-lexical mapping and feed the model directly with the lexical items. Whether phonological information as such flows into syntax is an interesting empirical question. The current algorithm allows some phonological information to pass, but the empirical data motivating this assumption is quite peripheral. `See https://youtu.be/-khLeU0K-90` for some thoughts on how to implement the lexical module.

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

The operations illustrated in (3) 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 (3) 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, in a manner first proposed by Phillips (1996). 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 (3)a can be ruled out on such grounds: *John furiously* is not an interpretable fragment. Verbal adverbs like *furiously* cannot be attached to proper names. Another problem 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 (3)a. This alternative can therefore be rejected on the grounds that the result is not consistent with the word order discovered from the input. If the default linearization algorithm proceeds recursively in a top-down left-right order, then each word must be merged to the *right edge* of the phrase structure, right edge referring to the top node and any of its right daughter node, recursively, as shown below.



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.[16] This leaves (3)(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:

(4) *Ranking*

a. [[~~John *furiously*], sleeps~~] (Eliminated, not consistent with linear order)

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 by backtracking and try solution (c). Every decision made during the parsing process is treated in the same way. This recursive procedure implements the derivational search function for the LPG: it considers every possible way of generating the particular input *John* ⊕ *sleeps* ⊕ *furiously* by applying Merge-1 and other grammatical operations discussed later. The mechanism can be illustrated with the help of a standard garden-path sentence such as (5).

---

[16] These assumptions depend of course on prior assumptions concerning linearization. The present model assumes a 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 in the context of L.

(5)

a. The horse raced past the barn.

b. The horse raced past the barn fell.

Reading (5)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.

Backtracking can be interpreted in one of two ways. If looked from the point of view of recognition grammar, its purpose is to consider all possible derivations available given the input and some grammar. The procedure will find 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.[17]

---

[17] It is not necessary 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. Aspects of realistic language comprehension are discussed in § 5.

The derivational search function that performs the ranking and recursive backtracking for the LPG theory is relatively complex since it incorporates additional processes such prosody, inflectional morphology, generation of the log file and other functions that are irrelevant for the core task described above. A simple entry point Python script that does only what was described above and which can be used as a starting point when constructing the derivational search function for an arbitrary recognition grammar using the abovementioned principles is provided below.[18] There is a YouTube video which discusses the construction of the derivational search function for this class of recognition grammars and its implementation.

```
206     def derivational_search_function(self, X, sentence, index):
207         if index == len(sentence):
208             self.process_final_output(sentence, X)
209         else:
210             Y = self.lexicon.lexical_retrieval(sentence[index])
211             for i in range(0, len(X.right_edge())):
212                 X_ = X.root().copy()
213                 right_edge_ = X_.right_edge()[i]
214                 new_X = right_edge_.MergeRight(Y)
215                 self.derivational_search_function(new_X, sentence, index + 1)
```

The function gets four input parameters: `self` (which we can ignore for the time being), `X` = the current phrase structure constructed on the basis of the input read so far, `sentence` = the input sentence as a list of phonological words and `index` = the pointer to the word in the input sentence currently being processed. If we have reached the end of the sentence (lines 207-8), we branch into another function (not shown here) which examines if the resulting phrase structure is well-formed and can be interpreted semantically. Otherwise we retrieve the lexical item corresponding to the input word (line 210) and try to fit it into each constituent at the right edge of X (lines 211-214) in some well-defined order determined by ranking (line 211). The recursive call is on line 215. There is only one operation in this grammar, Merge Right, which attaches all incoming lexical items to the right edge of the existing phrase structure. In a more realistic grammar the derivational search function might contain many additional operations (e.g., Agree, Move) that

[18] The whole script is available at
https://github.com/pajubrat/Templates/blob/main/template4.py

might apply at any point during the derivation and which must be included inside an additional loop (i.e. select any node at the right edge, then select any operation from a well-defined list). [19]

In the above function the execution branches into a separate function which handles the processing after all words have been consumed (line 208). This function will typically contain various well-definiteness checks that apply to complete phrase structure representations and various postsyntactic operations such as morphological adjustment and linearization. The operation Merge Right, on the other hand, applies to partial phrase structure representations. We therefore distinguish *cyclic operations* which apply potentially at each step during the derivation from *noncyclic operations* which apply only to complete phrase structures. Merge Right is a cyclic operation, because it applies potentially at every step. Operations such as Agree (modelling agreement) and Move (modelling displacement), both discussed below, may apply cyclically but they do not need to. Some operations could apply both cyclically and noncycically. All cyclic operations must be applied at least potentially at each stage of the derivation in order to implement a valid derivational search function.

Merge-1 can break constituency relations established at an earlier stage. This can be seen by looking at representations (2) and (3)c, repeated here as (6).

(6) 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 (7).

---

[19] This code example is discussed in `https://youtu.be/2JePzDkXAT4`.

(7) [John [[sleeps furiously] γ]

One consequence is that if we merge two words as sisters, we do not know if they maintain the same or any close structural relationship in the derivation's future. In (7), 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 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, a term we adopt likewise here.

The fact that constituency relations established during the left-to-right derivation are not guaranteed to hold in the derivation's future provides one reason to distinguish cyclic derivations from the noncyclic ones. The latter can be implemented under the assumption that the constituency relations are final, whereas this is not the case for the former. Hence operations like Agree (assume regular subject-verb agreement as an example) cannot be executed cyclically unless we know, based on some independent consideration, that the correct notions of subject and its verb have been established and will not change in the derivation's future. The same applies to Move: in many instances downward movement is not possible until the required grammatical elements have been processed and merged.[20]

The above examination presupposes that we have a notion of phrase structure at hand. So far we have assumed that constituents can have (immediate) left constituents and right constituents.

---

[20] Consider a typical filler-gap dependency such as *who did John ask __ to come to the party* where the interrogative *who* cannot be reconstructed into its thematic position (__) until at least the verb *ask* and its complement have been constructed, hence no straightforward cyclic algorithm is available. This is not to say that a cyclic reconstruction is impossible a priori; what is at stake is just a demonstration why developing one quickly becomes nontrivial.

These assumptions too must be formalized, and the formalization will require us to attend to several additional matters that have remained implicit so far, such as whether a constituent can have just one daughter constituent; whether each daughter constituents has a determined mother and, if so, whether each constituent can have only one mother; whether both lexical and complex constituent may have (lexical) features; what exactly it means to "copy" a constituent, and so on. The phrase structure formalism should ideally be defined in its own data-structure (class) and then presupposed in most or all syntactic operations. Formalizing any notion of phrase structure will provide a rigorous implementation of the notion of phrase structure. The intuition underlying LPG is that phrase structures are linear (motoric, semantic) plans arranged hierarchically.[21]

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 (8).

(8) 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

[21] [α β] is a composite plan made up of two plans α and β such as α is executed before β. Consequently, linear order and asymmetry are intrinsic properties of the phrase structure formalism and not added during some later stage of processing. In many bottom-up grammars available today, by contrast, the phrase structure is interpreted as a containment system: α and β are contained in the complex constituent [α β] hosting them.

**John admires to want him* or **John admires admires Mary* will be interpreted as grammatical i.e. generated by the grammar or accepted 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'. We can then check that the feature is satisfied after the phrase structure has been completed. The fact that *admire* has [+COMP:D] can also 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 (9), the test passes: the label of the selecting item matches with the label of the new word.

(9)

John	admires	Mary
↓	↓	↓
[John	[admires	Mary]]
	[+COMP:D]	[D]

Let us assume that [+COMP:L] means that the lexical item *requires* a complement of the type [L], [−COMP:L] means that it does not allow [L]. Type [L] is determined by a feature of the target, for example, by its lexical category. 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 *furiously*. What might be the lexical features associated with this item? There are three options in (9): (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 that a proper name cannot take an adverbial complement. This leaves us with (10)a-b.

(10)

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. Call such features *specifier selection features*. A feature [+SPEC:S] ("select clauses") favors solution (a), [+SPEC:V] favors solution (b)("select verbs"). What constitutes a specifier selection feature will guide the selection of a possible left sister during comprehension.[22] We can check that the specifier selection features are satisfied at the LF-interface.

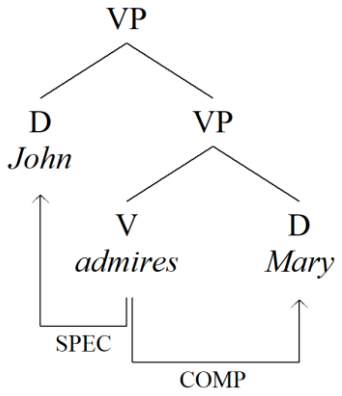
One limitation of the above scheme is that it allows each lexical item to specify only one complement and one specifier type. This is too restrictive. Many lexical items can have several types of complements and specifiers. One possibility is to allow each lexical item to have several selection features, each licensing one particular constituent type. Another is to allow each complement and specifier selection feature to contain a set of features which together provide an exhaustive list of selected features. Thus, [+COMP:A,B,…,C] means that the lexical item must have a complement of type [A], [B], …, or [C]; negative features can be dispensed with since anything that is not in these lists will be rejected. There are many ways of implementing selection of this type; what matters is that we adopt some scheme in order to begin the constructions of more realistic models as basis for further development.

Another missing assumption in all of the above is what exactly we mean by the notions "complement" and "specifier." This issue will be addressed below, but in general these notions

---

[22] 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.

should be defined as parts of the phrase structure formalism. For example, the notion of complement is closely related to sisterhood: when a verb such as *ask* selects its complement, this is usually interpreted to mean that it selects its (right) sister. Specifier refers intuitively to a left sister of the constituent containing both the lexical item and its complement:



The complement of *admires* is *Mary*, its right sister, whereas its specifier is *John*, the left sister of [VP *admires Mary*]. In this case the verb *admires* should have selection features [+COMP:D,…] and [+SPEC:D,…] where '…' contains a list of other complement types admitted by this element and which will jointly license the constituents shown above. Of course, nothing prevents one from proposing different definitions for the notions of complement and specifier; final definitions will be provided in Section 4.6.[23]

The simplest starting point is to assume that the features are strings that can be provided easily in an external file and processed by relying on build-in operations provided by most higher-level programming languages available today. The algorithm will then contain corresponding code

[23] In some of our examples we have labeled complex constituents by using linguistic symbols such as S and VP, referring to "sentences" and "verb phrases," respectively. These properties have not been introduced thus far: all complex constituents were assumed to be asymmetric combinations of their constituents, therefore they do not have labels such as S or VP. Labels will be added formally to the model in Section 4.5. But notice that as soon as we begin to introduce more realistic selection features, we will end up with a situation in which we must select for complex constituents and not just for lexical items such as *John* and *Mary*. Selection for complex constituents can be implemented by using the same selection features; all we need is a way to provide complex constituents with labels (e.g., VP, S) that can be selected.

which parses the strings and provides them with "implicit definitions," thus some definite role in the overall system. An alternative is to define a notion of linguistic feature in its own class, in which case the class would provide all the parsing functions required elsewhere.[24] The string implementation is superior since it allows one the provide the features in an external text file, where they can be edited by using standard text editors.

4.4 Phases and left branches

In this section we will make a brief comment concerning the ordering of the computational operations involved in the left-to-right cycle. Consider the derivation of (11).

(11) John's mother admires Mary.

↓ ↓ ↓ ↓

[$_S$[$_{DP}$ John's mother] [$_{VP}$ admires Mary]]

After the finite verb has been merged with the DP *John's mother*, no future operation can affect the internal structure of that DP. Merge-1 is always to the right edge. All left branches therefore become *phases*: the derivation can forget them inside that particular parsing path.[25] It follows that if no future operation is able to affect a left branch inside the that parsing path, which is indeed often the case, 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 in this way. 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

---

[24] What is at stake is the ultimate nature of linguistic features: are they simple patterns, perhaps something processed by simple first-order Markov operations, or something more complex, perhaps a representational system of their own? What is the relationship between linguistic feature and phrase structure?

[25] 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).

leads into an unrepairable left branch can be either filtered out as unusable or at the very least be ranked lower in the parsing search. We can formulate the condition tentatively as (12).

(12) *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 at hand, 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 (12), although it leaves it open what these "operations" are. We can express the same intuition by saying that left branches are calculated cyclically, during the consumption of the words from the input.

The left branch condition (12) is an "architecture" property of the LPG in the sense that it follows from the fundamental assumptions of the model and does not have to be stipulated as an independent principle. It has certain important and interesting empirical consequences. For one, it explain why left branches behave like closed islands with respect to several operations such as movement (one half of the so-called CED constraints; (Huang, 1982)). This property is illustrated in (13).

(13) a. Who did John admire __?

b. *Whose did [__ brother] admire John?

Both (13)a and (b) are formed by positioning the interrogative pronoun to the first position of the clause, but only (a) is grammatical. This is because in (13)b the interrogative has been displaced from within the subject that constitutes a left branch in the sense of (12). These data follow from (12), which treats the left branch as a closed environment with respect to whatever operation is responsible for the interrogative word order.

4.5 Labeling

In this section we focus on labeling of complex constituents. Suppose we reverse the arguments in (11) 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.

One solution that is clearly wrong would be to assume that complex constituents can have their own independent labels. Under this scheme, a constituent like *John's mother* could be a verb phrase, adjective phrase or a non-finite clause. It is clear that labels of complex constituents depend on their contents. *John's mother* cannot be a non-finite clause, for example. We can model the dependency between the labels of complex constituents and their contents 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,[26] 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, where "closest" refers to 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 of the whole complex constituents. Example (16) provides some basic examples of how (15) works. Label L for complex constituents is provided by notation $[_{LP} \ldots]$.[27]

(16) Structure ~ label

a. $[_{VP}$ *admires Mary*] ~ left, verb;

b. $[_{PP}$ *from Mary*] ~ left, preposition;

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

d. $[_{DP}$ *the man*] ~ left, primitive D;

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

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

The definition refers to the notion of complex constituent. *Complex constituent* is defined as a constituent that has both the left and right constituent; if a constituent is not complex, it is *primitive*.[28] Consider the derivation of (2), repeated here as (17).

---

[26] The notion of adjunct has not been defined so far and will be discussed later; a phrase structure geometry that does not have a separate category of adjuncts can simply ignore for the movement the conditions referring to them.

[27] The rule is defined in this way because this definition generates correct and/or plausible results; the matter is empirical, however, and it is easy to experiment with different alternatives. For example, these assumptions rule out configuration $[_{\beta P}\ \alpha^0\ \beta]$ 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.

[28] A constituent that has only the left or right constituent, but not both, will be primitive according to this definition. A constituent that has no daughters is called *terminal*.

(17) 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, (17) is a sentence or verb phrase, not a noun phrase. For example, it cannot be selected by transitive verbs (**John admires* [*John sleeps*]). The problem is that we have tacitly assumed that *John* must correspond to a primitive constituent, whereas all we know is that it appears as a monomorphemic word at the level of surface expressions. We are therefore free to assume that *John* corresponds to a complex constituent, say with the structure [$_{DP}$ D N]. Let us assume, furthermore, that this information comes from the lexicon, where proper names are decomposed into D + N structures. The structure of (17) is therefore (18).

(18) John ⊕ sleeps.

↓ ↓

D N sleeps

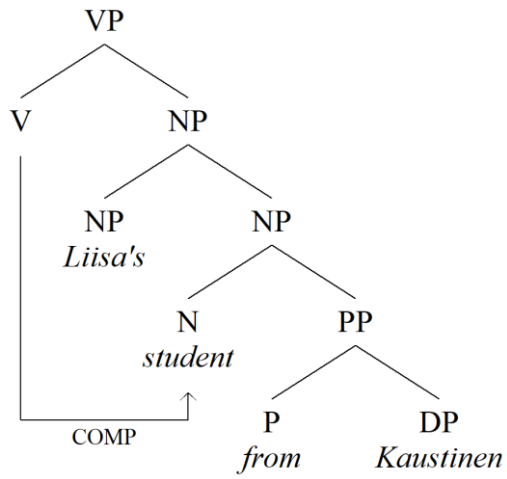
↓ ↓ ↓

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

This will provide V as label, much more realistic result that if it (18) were analyzed as NP. The structure of the lexicon is examined in detail in § 4.9.2, at this point it is important to note that we have not assumed (and will not assume) that all phonological words are mapped into primitive lexical items. However, whether some word is or isn't decomposed is crucial for labeling, because the labeling algorithm elucidated above is sensitive to the distinction between primitive and complex constituents.

We can now reinterpret the system of specifier and complement selection introduced in Section 4.3. Instead of selecting the constituents themselves, we select their *heads*. For example, if we

assume that transitive verbs select regular noun phrase (NP) complements, as one might assume in an articleless language like Finnish, we can have a feature [+COMP:N…] which implements the required dependency



where V selects N and, because it now targets the head of the complement, selects NP arguments independent of whatever other constituents and words (here the possessor and the preposition phrase) they might contain. In a theory which assumes that all regular arguments are DPs, V would select D will be the head. If the theory allows for both, then V could have feature [+COMP:D,N,…].

Because the labels for complex constituents are provided by a separate labeling algorithm instead of being intrinsic properties of the constituents themselves, the system is based on the *bare phrase structure* model (e.g., Chomsky, 2008, 2013; Chomsky et al., 2019) while assuming inherent left-right asymmetry. The term denotes phrase structure formalisms in which complex constituents do not have intrinsic labels; rather, they are bare combinations of their constituents and nothing more. It is possible to assume even less and remove the asymmetry, yielding a symmetric set-theoretical phrase structure formalism with unordered constituents {α, β} (e.g., Chomsky, 2013, 2015, 2021; Chomsky et al., 2019; Marcolli, Berwick, et al., 2023; Marcolli, Chomsky, et al., 2023). Everything stated above could be implemented in principle by adopting a set-theoretical phrase structure, although the labeling algorithm, and hence selection, cannot then rely on asymmetry.

LPG on the other hand is premised on the assumption that the phrase structure relies on linear order as one of the fundamental and basic properties of language and linguistic communication.[29]

4.6 External and internal search paths

Constituents making up phrase structure representations can enter into several types of dependency relationships. Some of these dependencies are too nonlocal to be modelled by relying on lexical features and on the notions of complement and specifier. To illustrate, consider the following pair of Finnish sentences.

(19) a. Pekka ei löytänyt *Merja-n/ Merja-a.
Pekka not find Merja-acc Merja-par
'Pekka did not find Merja.'

b. Pekka on löytänyt Merja-n/ *Merja-a.
Pekka has find 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.[30] Both sentences, when generated by Merge-1 from the input, generate a representation approximated in (20). The dependency between the negation and the direct object is shown by the line under the sentence.

---

[29] To be more specific, it is assumed in LPG that hierarchical phrase structure representations utilized by human language faculty constitute an evolutionary extension of much older linear processing that creates, for example, motoric sequences both in the case of human and nonhuman animals. Human evolution provided our ancestors an access to hierarchical linear sequences – phrase structure constituents. Regardless of the evolutionary history, language and linguistic processing is linear (an assumption that is by no means new, see (Kayne, 1994)). For Chomsky, the crucial evolutionary step was the emergence of the containment relation formalized by the set-theoretical bare phrase structure (Chomsky, 2021). In this theory, linear order is secondary.

[30] The empirical facts are more complex, but they are irrelevant to the main point.

(20) Pekka ei/on ihaillut Merja-a.

[Pekka [not/is [admire Merja-par]]]

This relationship cannot be described by local selection of the type described in §4.3. The case form at the case assignee, or some underlying feature, must enter into a "compatibility check" with another element, the case assigner (or feature thereof).

To model dependencies of this and other types (discussed further below), we assume that any constituent may examine its external environment or context ('what elements appear in its surroundings') and its internal constitution ('what elements appear inside it'). Both notions are implemented by search algorithms we call *external search* and *internal search*, respectively.

External search is defined as follows. Suppose α is the target element executing external search; then

(21) *External search path*

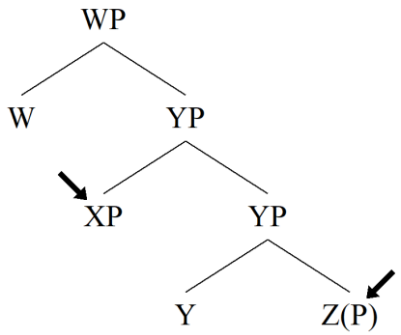
the external search path from α contains (i) all constituents that dominate α (called the *spine*) plus their (ii) immediate daughters that do not belong to (i)(called the *branches*).[31]

The path itself is the collection of constituents visited by the search algorithm. Intuitively the case forms in (19) are calculated by matching them with relevant elements in the search paths. For example, the accusative forms are rejected if a negative word is encountered by external search which looks "upwards" from the case-marked element. This procedure creates a grammatical dependency between the two elements, but what matters for the LPG is not so much the

[31] This notion has a somewhat long history in the work presented here. It was first invoked in the author's 2003 thesis and was there called the "feature vector." It was generalized and re-adopted later to model several grammatical mechanisms in the present framework, including control, binding and case assignment. A further impetus for the later studies came from the connectness approach of (Kayne, 1983, 1984)

dependency as an abstract notion but the actual search algorithm that implements it – the former exists as an effect of the latter and has no independent existence.

Case assignment is not the only domain that relies on external search: it is used to model almost all dependencies, even the local ones. For example, the notion of path provides reductive definitions for complement and specifier provisionally introduced in §4.3. Suppose each external path can have a *domain* which limits its upward scope, and one of such domain is the maximal projection from the target head α. Then with domain = maximal we define the complement of α to be the first branch in the path (possibly also the first *right* constituent), and the local specifier to be the first left branch.[32] Consider the following phrase structure.



The *spine* of the path from Y contains YP, second YP and WP, while the *branches* are Z(P), XP and W. If we limit the domain to maximal projection, the path from Y contains Z(P) and XP – the complement and the specifier, respectively. Because the domain is restricted to maximal projection, Y will not become the complement or specifier of Z, since YP is not a maximal projection from Z, while Z will still be the complement of Y. All left heads are excluded from the class of specifiers (from Y) for the same reason: no W can be be a specifier of Y under $[_{\alpha P}\ldots W\ldots[_{YP}\ \ldots Y\ldots]]$ since W projects. It follows that all specifiers must be phrases.

[32] Alternatively, we could define the complement to be the first constituent, the specifier the second, but this model requires counting and was rejected.

A detailed linguistic examination of how the notion of external search is used in LPG to reduce linguistic dependencies across-the-board is beyond the scope of this document. The essence of the hypothesis is easy to state, however: LPG reduces grammatical dependencies, especially the nonlocal ones, to a search algorithm that then becomes one of the most fundamental mechanisms of the theory.

One question that could arise here is why to rely on search algorithms instead of more static dependency relations formulated, say, by means of well-formedness conditions. The reason is because any implementation of such static notions, especially when they are not restricted to any local domain in principle, will ultimately have to rely on search, some mechanical (usually recursive) way of exploring the phrase structure. The algorithm cannot "eye-ball" dependencies from the phrase structure or in any other way detect their presence unless it is able to execute search. The question is not whether the search algorithm must be recursive or something less powerful; the issue is that the the implementation of dependencies of this type must rely on a mechanism for systematically exploring the phrase structure, and thus it was assumed that the search algorithm itself – and not the static configurations that it generates – will form the basis of the dependencies. The neuronal implementation remains an interesting question. My own hypothesis is that the search paths correspond to a notion of syntactic working memory that the search algorithms define. Under this interpretation, search = exploration of syntactic working memory.

In addition to external search, there must also exist *internal search* which looks inside any constituent. The labelling algorithm provided in Section 4.5 can be reinterpreted as a special case of internal search, where the target is the first (most prominent) primitive head inside the target constituent. Internal search can then be used to model operations like agreement (Agree) and many others. Internal search will be examined in detail later.

4.7 Adjunct attachment

Adjuncts, such as adverbials and adjunct PPs, present a challenge for the incremental and cyclic parser. The topic is complex one, thus we can only hope to scratch the surface and present the general idea while the details are filled in later sections. Consider, to illustrate the problem, the data in (22).

(22) (Finnish)

a. *Ilmeisesti* Pekka ihailee Merja-a.
apparently Pekka.nom admires Merja-par
'Probably Pekka admires Merja.'

b. Pekka *ilmeisesti* ihailee Merja-a.
Pekka apparently admires Merja-par

c. Pekka ihailee *ilmeisesti* Merja-a.
Pekka admires apparently Merja-par

d. ?Pekka ihailee Merja-a *ilmeisesti.*
Pekka admires Merja-par 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: the position of the adverb is essentially unpredictable. The same pattern applies to relative clauses (23)a, adjectives (23)b and preposition (23)c phrases.

(23) a. Se mies [jonka tapasit eilen] ihailee Merja-a
that man who met.2sg yesterday admires Merja-par
'That man, who you met yesterday, admires Merja.'

b. se pieni punainen talo
that small red house
'that small red house'

c. matka kohti Pariisi-a

trip towards Paris-par

'a trip towards Paris'

Before attempting to find a useful model for adjunction, it is important to survey a large sample of adjunct phenomena in the language(s) of interest. The relevant dataset is too large and heterogeneous to convey here in any useful way: just consider relative clauses, appositive relatives, adjectives, adverbs, preposition phrases and DP-adverbs. A theory of adjunction should handle them all or, if not, then there must exist several theories that apply to different construction classes, an equally nontrivial scenario.

In the LPG adjuncts are assumed to be geometrical constituents of the phrase structure, but they have a special property in that they are stored inside a parallel syntactic working memory making them invisible for sisterhood, labeling, external/internal paths and selection in the primary working memory (making them invisible for paths automatically generalizes invisibility to specifiers, complements, sisterhood and labeling algorithm). In a sense they increase the 'dimensionality' of the phrase structure by providing it with invisible satellite constituents. The idea is illustrated in the following figure.

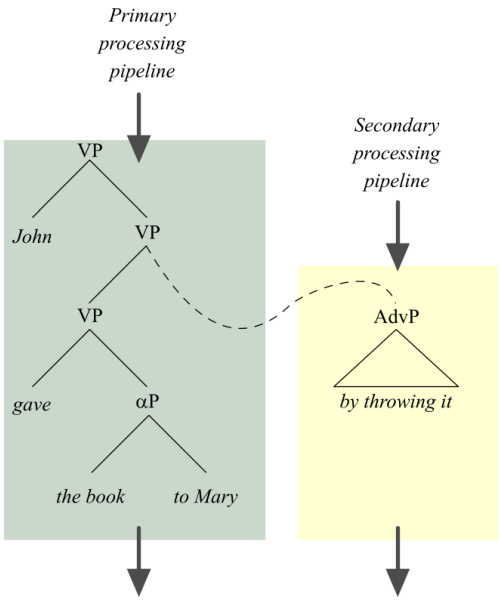


Figure. Adjuncts are geometrical constituents that are pulled out of the primary working memory and are processed inside separate processing pipeline. We can imagine that the VP is made up of two segments.

Thus, the labeling algorithm, as specified in §4.5, ignores adjuncts. As a consequence, the label of (24) 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.

(24) [$_{VP}$ John [$_{VP}$[$_{VP}$ admires Mary] ⟨$_{AdvP}$ furiously⟩]]

Consider (25) next.

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

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.[33] The fact that adjuncts, like the adverbial *furiously* here, are optional follows from the fact that they are excluded from selection and labelling.

While this is the basic idea, there are many details that we leave here unattended, such as how adjuncts are recognized in the input, how they are created once recognized and how they are reconstructed if they appear in a wrong position in the input. One additional detail merits discussion. It follows from these assumptions that adjuncts could be merged anywhere, which is not correct. For this reason each adverbial (its 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,

---

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

constitutes a satellite 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) relates to the corresponding head by an external path. Conditions (i-ii) have slightly different content and are applied in different circumstances.[34]

(26) 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 β_FP or is sister of β_F;

b. α can be linked with β_F by an external path (21).

β_F is a head with [F]. Condition (26)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. This is also the notion that one should being with, since there is not much doubt that something like this must be part of the human language faculty. Condition (26)b licenses adjuncts in lower positions than what is indicated by their tail-features. This condition is currently used to model case-based adjunction, but the matter is too controversial to merit discussion here; condition (26)a is the core case.

The definition of adjunction licenses both left and right adjuncts. Left adjuncts are linearized to the left of their host head (e.g., adjectives in Finnish), right adjuncts to right (e.g., relative clauses, adverbials in Finnish). Left adjuncts are very similar to regular left phrases, because left phrases, whether adjuncts or not, are ignored by the labeling algorithm.[35] Left phrases and left adjuncts are not identical, of course, since the former can be selected while the latter can't; they generate

---

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

[35] Notice that the label of [α β] will be β if α is complex, α does not have to be an adjunct; if α is an adjunct, the situation will be the same.

different sisterhood dependencies. They differ also behaviorally. One differences is that left adjuncts can be stacked while left phrases (specifiers) 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 another path into the tertiary structure. This is of course normal in language where an adverbial phrase may contain further modifiers. There is no in principle limit on the number of adjunction dimensions any given structure can have.

4.8 Transfer

4.8.1 Introduction

After the phrase structure has been composed by cyclic derivation from the input, 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. These operations are also called *reconstruction*: they reconstruct the expression for semantic interpretation and further processing.

4.8.2 General comments

Let us first consider transfer and reconstruction in a more general way. 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 cyclic Merge-1 operations the constituent ends up in a position where it passes all LF interface tests and can be interpreted semantically together with its syntactic and semantic context. 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 (27) illustrates a simplified sentence where this is the case.

(27) John admires Mary

↓ ↓ ↓

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

A legitimate output is reached by merging the input words into the phrase structure as it is being created by the left-to-right cycle: the operation brings all arguments inside the verb phrase where their correct thematic roles are determined. Example (28) shows a variation in Finnish where this is no longer the case.

(28) Ketä Pekka ihailee? (Finnish)

who.par Pekka.nom admires

'Who does Pekka admire?'

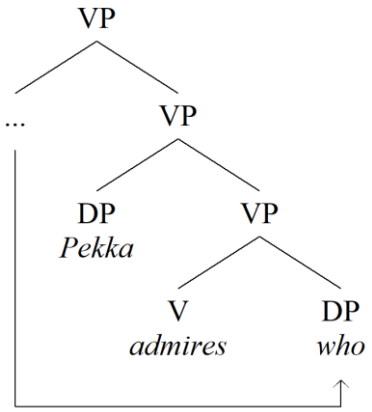
The model generates the following first-pass parse for this sentence (in pseudo-English for simplicity):

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

This solution violates two selection rules. First, there are two specifiers at the left edge, *who* and *Pekka*. The second problem is that *admire* lacks a complement, the target of admiration. Even if the parser backtracks, it cannot find a legitimate solution.[36] The two violations are not independent, 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 considering

[36] A candidate structure such as [$_{VP}$ [$_{DP}$ *who Pekka*] *admires*] is also illegitimate. In principle we could feed (29) 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*.

what would fix both problems: movement of the interrogative pronoun from the left edge to the complement position of the verb, an operation like *who Pekka admires* → *Pekka admires who*:



Dislocation is a general feature of natural language. In Finnish almost all word order variations of the referential arguments of a finite clause are possible, yet these variations do not affect the thematic roles of the arguments (30).[37]

(30) a. Pekka ihailee Merja-a. (SVO)
Pekka.nom admires Merja-par
'Pekka admires Merja (Pekka = agent, Merja = patient).'

b. Merja-a ihailee Pekka. (OVS)
Merja-par admires Pekka.nom
'Pekka admires Merja (Pekka = agent, Merja = patient).'

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 only stylistic differences. 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.

---

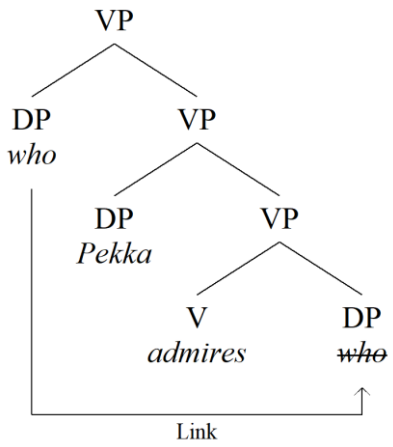
[37] Finnish speakers use overt case forms, here the nominative and partitive, to reconstruct the thematic roles (Brattico, 2020, 2023c).

Almost all reconstruction operations are based on the same general 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 an offending constituent, it will attempt reconstruction, where a legitimate position is searched for. 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 at its surface position. In that case, some later operation may 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.

The observation that we have to "move" or "displace" constituents generates several additional questions that needs to be answered before an operation of this type can be implemented. One possibility is that the moved element disappears from its input position and re-appears in its reconstructed position. This model does not work in those cases where movement has some semantic or pragmatic function, as is usually (though not always) the case. A better approach is to copy the element, so that it appears in two or more positions in the final output phrase structure. It is not possible, however, simply to duplicate phrases and heads in the structure since that would confuse copies and repetitions: *John met John* does not usually let alone necessarily mean 'John met himself', rather it refers to two separate persons that happen to have the same name, whereas copying in the sense elucidated above does not multiple referents. Copying creates two instances of the same element, whereas repetition creates two distinct tokens of the same type.

To understand what copying is and how it differs from repetitions, let us consider the lexicon. The lexicon contains a permanent storage of lexical items (defined by lexical entries, lexical knowledge) that must point to further conceptual representations, encyclopedic knowledge and whatever other semantic information is associated with each word/concept. We assume that the storage is permanent: it is not destroyed except in the case of severe neuronal pathologies. At the same time, lexical items must be able to occur in the phrase structure representations as terminal elements. In that capacity, they are not permanent: they come and go as sentences are processed.

So we have a permanent storage from which elements are *copied* into the transient linguistic representations. Furthermore, most of the semantic attributes associated with each lexical item/concept will not get copied into the transient syntactic representations; rather, the copy and the original must maintain a link. The LPG generalizes this model to reconstruction: when a constituent is reconstructed, it is copied much like lexical items are copied, and the copy has a pointer to the original that allows the semantic system to reconstruct their relation. Then during processing copied constituents are ignored and replaced by the source constituent indicated by the copy-link, much like what happens during lexical retrieval when lexical items allow us to retrieve all semantic attributes and knowledge associated with them. Because LPG reconstructs expressions from inputs, copy-come to links point *from* the surface elements *to* the base positions:



The interrogative pronoun at the base position contains another link which associates it with features and notions in the semantic system.

One of the problems discussed in the linguist literature is why language makes use of displaced elements. In the case of the lexicon, the reason is obvious: we do not want to destroy lexical elements when they are in linguistic communication. Perhaps the same logic applies to the above representations, in other words the system does not want to destroy (remove, eliminate) the original constituent since this would distort meaning, leaving the transitive verb without an object in this case. But why is it copied to the higher position? In the LPG it is assumed that the dislocation in this particular example serves the function of expressing the fact that the sentence

is an interrogative and that its scope is the whole clause ‘which x: Pekka admires x’. By using this device we can separate the following two sentences from each other:

(31) a. Who does Pekka admire __?

‘which x: Pekka admires x’

b. Who did Merja believe that Pekka admires __?

‘which x: Merja believes that Pekka admires x’

More generally, the displacement operation serves a linguistic function of expressing one part of the meaning of the sentence – the scope of the question. This is reflected in a later processing step when the semantic system reads the scope information from the final phrase structure representation at the LF-interface. The same reasoning applies to Finnish free word order. Although the various word orders leave the thematic roles intact, they express information structural notions such as topic and focus. We can understand this phenomenon if we assume that the syntax-semantic interface contains an interpretative function which maps word orders (or changes in canonical word orders) into information structural notions such as topic and focus. A language that is able to reconstruct the relevant word orders will then come to possess an extra mechanisms for communicating information structure.

4.8.3 Cyclic and noncyclic transfer

In all of the previous versions of the LPG (versions 1-19) all reconstruction was noncyclic: it applied to finalized left branches and final phrase structure representations (see §4.4).[38] The system discussed in the previous section was also noncyclic, though the assumption was made without explicit comment: displacement was applied to complete phrase structure representations. The only cyclic (cyclic = perform the operation for partial phrase structures, or “as soon as

[38] Recall that left branches are transferred as independent phases. This does not count as cyclic transfer, since the operations are applied to what amounts to completed phrase structure representations.

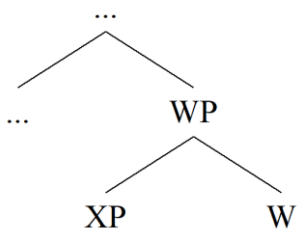
possible") operation was Merge-1 and the creation of the derivational search space by recursive backtracking (filtering and ranking), but this is unavoidable in any incremental model.

There were, and are, several reasons which suggest that some transfer operations must be *noncyclic*. One reason is the existence of the free word order phenomenon, or scrambling, in which a thematic argument can occur almost at any position in the finite clause. Such arguments cannot be reconstructed cyclically, if only because in many cases the thematic base position for the argument is created before the argument itself has been processed, thus there is nothing to reconstruct when the final base position is created. The fact that thematic arguments behave in this manner also means that any attempt at relying solely on cyclic agreement processing (Agree) will likewise fail: the displaced argument must first get reconstructed, and only then trigger agreement. We examine some examples of these properties below. Long-distance filler-gap dependencies pose similar problems. Finally, Finnish long head movement behaves in a way that prevents any attempts at performing universal cyclic head reconstruction (Brattico, 2022). These were the three main reasons why all reconstruction was first performed noncycically. It does not follow, however, that *all* reconstruction must be noncyclic. Indeed, some later work provided evidence suggesting that some reconstruction is cyclic, thus it can and must be performed as soon as possible.

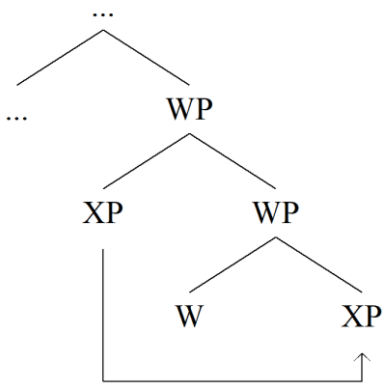
There were two main reasons why cyclic repair/reconstruction was introduced to the model from version 20.0. One reason is theory-internal, and has to do with the fact that if all head reconstruction is postponed until noncyclic transfer, several target positions for operations such as adjunct attachment remain invisible during the cyclic portion of the derivation. This was always a problem that resurfaced in many studies.[39] If head reconstruction is, at least in part, a cyclic operation, then the positions become available earlier, providing more right edge positions for Merge-1. But the more interesting reason is that cyclic reconstruction provides a natural

[39] The problem was fixed by adjunct reconstruction, but this strategy produced nonoptimal results in some cases.

explanation of why certain operations are local: the nonlocal structure simply does not exist at the stage the operation applies. In the earlier model these locality properties were stipulated by locality axioms. To see this, suppose the model has derive the following structure and determines that XP requires repair/reconstruction.



The only cyclic repair "position" available here is the following, since no further structure has been assembled.



The local nature of both A-reconstruction (A-chains) and local HMC-compliant head reconstruction is captured in this way in the current LPG. The system will perform *some* repair operations as soon as the relevant objects are in the phrase structure. Other reconstruction operations, such as scrambling reconstruction, are performed noncyclically.

4.8.4 Ā-chains

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

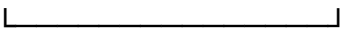
(32) Ketä Pekka ihailee?

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. The reconstruction algorithm searches for the first gap for this element where it can generate a legitimate position and reconstructs (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 get fixed: there are no longer two specifiers at the left end of the sentence, and the complement is no longer missing. The scope of the question can be recovered from the two constituents.

The search algorithm proceeds from the deviant element downstream (by using internal search) 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 *internal search* (§ 4.6, 4.8.8) which targets the sister of the operator, in this case the highest VP-node. Internal search "looks inside" that node. The element is copied to the first position in which (i) it can be selected, (ii) is not occupied by another legitimate 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. A copy-link will occur at the higher element.

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. While it is possible to say *John asked who John admires*, **John claimed who John admires* is ungrammatical. The bare VP representation does not yet represent this fact, because it was always interpreted (implicitly, as it were) as a declarative proposition. How do we distinguish interrogatives from declarative clauses? Interrogativization is represented by a wh-feature that must be part of the highest head in the sentence, so that it can be selected "from above" 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).[40]

(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]

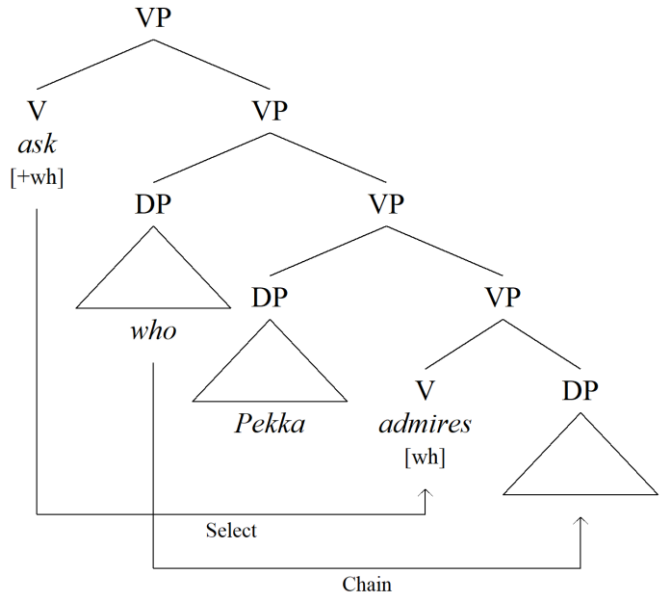
b. John asks/claims + [CP+$_{wh}$ ~~who~~ C$_{wh}$ [Pekka [admires who]]]

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. It might be that it is the presence of phonologically null heads which motivates the displacement in the first place. An alternative is to try to generate the wh-feature into the highest head that already exists, thus in the above case the verb head of the verb phrase would contain the wh-feature, licensing appropriate selection from above (36).

---

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

(36)



Because the current LPG does not accept double-specifier constructions like (36), it will generate an extra head between the two arguments. If the space between the two constituents is filled by something, as is the case with English interrogatives with Aux-inversion (37), then that element can receive the wh-feature.

(37) Who does$_{wh}$ Pekka admire __?

A third option is to always project a dedicated head such as C to represent clause typing (declarative, imperative, interrogative).

As pointed out earlier, the dislocated phrase represents the propositional scope of the question. The interpretation is one in which the speaker is asking for the identity of the object of admiration ('which x: Pekka admires x'). 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.

(38) ~~who$_{wh}$~~ C$_{wh,fin}$ 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. They have a syntactic side, because the dislocated deviant element must be reconstructed by a syntactic operation that takes place during transfer, by internal 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.[41, 42]

Some Ā-chains could be reconstructed cyclically, but in the current implementation the operation is noncyclic. A cyclic version was implemented and applied successfully to cases in which the interrogative was reconstructed into a locally created specifier position, but was later removed since it created problems that could not be solved elegantly without the help of several ad hoc assumptions. Cyclic reconstruction into a complement position seems impossible, since the operation cannot be executed until we know what type of words (if any) are still left in the input.

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

---

[41] 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).

[42] 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.

the scope of the question. The operation is called Move or Internal Merge. Reconstruction is its mirror image, a reverse-engineered Move operation; they describe the same object, a directionless *chain* that is constituted by the two (or more) members in two (or more) syntactic positions.

4.8.5 A-chains

In addition to Ā-chains discussed in the previous section many languages exhibit another type of phrasal reconstruction, A-chains. Let us again illustrate the issue by using a simple example. The English preverbal subject position can be occupied by both the agent and patient (39).

(39) 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 (39)b, because *Mary* would be in the agent position, *John* the patient (if present). We conclude on the basis of (39)a–b that the English preverbal subject position is *not* a thematic position. It can be occupied by both agents (39)a and patients (39)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 tensed auxiliary verb *was* present in the example (39)b. Note that both the bare finite verb (a) and the aux-verb construction (b) involve a 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 that may be expressed either by combining it with the verb (a) or by the auxiliary verb (b). We can depict the situation by using the schema (40).

(40) [$_{TP}$ 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, which is called *head movement/reconstruction*, or *head chain*, will be discussed later. 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 by saying that 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 (41).

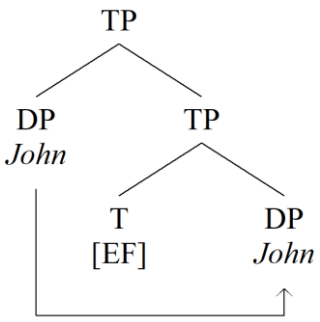
(41) [TP ~~John~~1 [TP T [VP John1 [VP admire Mary.]]]

This operation is 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, which is SpecVP. The fact that SpecTP is a nonthematic position is marked in the current implementation by an edge feature feature [EF], discussed later in more detail but corresponding to the traditional EPP-feature. 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.

Representation (41) is already very close to how the real model represents canonical transitive or intransitive clauses. We assume that they are composed 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. The two parts have different semantic roles: VP denotes an abstract configuration – perhaps some type of core proposition – with a predicate or relation together with the participants, while T creates a spatiotemporal *event* from the VP and denotes it in the global

semantic space.[43] Because the VP creates an abstract proposition, it does not have to denote an event itself; it can be nominalized as well.

A-chains can be created both cyclically and noncyclically. In the current implementation they are created cyclically: as soon as T with [EF] is merged-1, the argument (if any) is moved to a new position:



An alternative is to create A-chains noncyclically and insert the element into the vacant position between T and V. The latter model requires a separate locality axioms which block long-distance A-chains, whereas the cyclic model can eliminate such stipulations (no other positions exist at the time of reconstruction). Also successive-cyclicity follows from the cyclic model: if another EF-head is merged, the phrase can perform the same operation. This can continue until it reaches a thematic base position such as SpecVP.

4.8.6 Head reconstruction

Up to this point we have assumed that words and even single morphemes can be mapped into several heads. The operation that implements the relevant mappings is called *head reconstruction* (also head movement, head chain). Let us consider again (42).

(42) John admires Mary.

---

[43] That is, TP projects an event object into the global semantic space.

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. 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. 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 (43).

(43) 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 (43) is selected because T can select a VP complement. If the operation is performed cyclically, there is no other choice than the closest position; if noncyclically, a separate locality axioms must apply.

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 up to this point is a situation where the complex constituent has only one daughter constituent. This configuration creates nonphrasal complex constituents: the complex heads referred to above. This yields $\alpha(\beta)^0 = [_\alpha\ \beta]$ where α is a constituent that has only one 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.[44] Complex words of this type are notated in the phrase structure by writing the single constituents into vertical stacks.

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...]]]? Earlier versions of LPG modelled complex words in this way. The reason this approach 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 (44).

---

[44] 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 $\alpha \oplus \beta$. Each element in the sequence must be atomic and map into one sensorimotoric program.

(44) 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 (44) by reconstructing complex heads syntactically. Head reconstruction required by (44) has to be noncyclic; local head reconstruction can be implemented cyclically and noncyclically (local head reconstruct is cyclic, thus words are expanded into phrases as soon as they are created by the derivation unless the operation is blocked by the fact that the word itself is an operator, as it is in (44), in which case it is postponed to the noncyclic part).[45]

4.8.7 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). To illustrate, consider the pair of expressions in (45) and their canonical derivations.

(45)

a. Pekka käski meidän ihailla Merja-a.

---

[45] If we assume 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. It must be possible for speakers to perceive words as composed out of individual morphemes. In other words, there must be a process which can decompose phonological words into their constituent 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.

Pekka.nom asked we.gen to.admire Merja-par

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

'Pekka asked us to admire Merja.'

b. Merja-a 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. 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 similar to the distribution of adverbials in Finnish, one can argue that richly case marked thematic arguments can be promoted into adjuncts. Case forms are morphological reflexes of tail-head features (§4.7). If the condition is not checked by the position of an argument in the input, then the argument is treated as an adjunct and reconstruction into a position in which the condition is satisfied. In this way, the inversed subject and object can find their ways to the canonical LF-positions (46).

(46) [⟨Merja-a⟩$_2$ T/fin [__$_1$ [käski [meidän [ihailla [__$_2$ ⟨Pekka⟩$_1$]]]]

Merja-par asked we.gen to.admire Pekka.nom

'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. Scrambling reconstruction must be noncyclic operation and as such applies only to finished phrase structures.

4.8.8 Internal search

Let us consider once more a standard interrogative sentences such as (47).

(47) 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 *internal search* (also "minimal search"), first discussed in §4.6. The gap position is located by descending downwards through the phrase structure from the position of the reconstructed element and detecting the first legitimate thematic position. More formally, at each step of the internal search procedure 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.[46] 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 (47). To reach the reconstruction site $\_\__1$, the search algorithm must avoid going into the subject A and into the temporal adverbial B. Internal search follows the projectional spine of the sentence and ignores both left specifiers/adjuncts and right adjuncts (there are no "right specifiers" in the model).[47]

The same internal search is used to find the head/label of any phrase, to implement agreement (next section), determine whether a given phrase is an operator, and indeed in many other operations. We can think of it as a way of determining the internal constitution of a phrase structure constituent. External search, discussed in §4.6, looks at the external context of the constituent. External and internal search are very similar, though in the current model they haven't yet been unified under just one search algorithm – a very desirable goal, if possible. They are implemented as literal search algorithms: external search looks into upwards direction (via

---

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

[47] There is a possible deeper motivation for internal 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 internal search coincides with the contents of the current syntactic working memory at any point in the derivation.

dominance relations) by utilizing a linear (first-order Markovian) loop, while internal search uses recursion since it is in some cases able to branch. The internal search illustrated above does not branch but could be implemented by a linear loop; however, this seems not to represent the general case. While each constituent can have only one mother, it may have at most two constituents, hence branching and recursion when looking inside.

4.8.9 AgreeLF

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 (48).

(48) a. John admires Mary

b. *John admire Mary

c. *They admires Mary

The third person features of the subject are cross-referenced in the finite verb and expressed by means of the singular third person suffix *-s*. The term agreement or phi-agreement refers to a phenomenon where the gender, number or person features (collectively called phi- or φ-features in this document) of one element, typically a DP, covary with the same features on another element, typically a verb, as in (48).

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 human individual'. We can see this effect for example when we note that the pronoun *him* in sentence *Mary said that John admires him* cannot be coreferential with *Mary*. A predicate, on the other hand, must be linked with an argument that has interpretable phi-features, which together form a semantically well-formed package of linguistic information, predication. To model this asymmetry, a predicate has *unvalued* phi-features, denoted by [φ_]. An unvalued phi-feature is simply an 'incomplete' phi-feature, such as number, that has not been specified for any particular value (singular, plural). The value will be (usually) provided by the argument with which the predicate agrees with. This called *valuation*. The starting point is (49).

(49) 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 *AgreeLF* (50).

(50) Mary [admires John]

[PHI:NUM:SG] [PHI:NUM:SG]

[PHI:PER:3] [PHI:PER:3]

[PHI:DET:DEF] [PHI:DET:DET]

└── AgreeLF ──┘

As a consequence, the unvalued features disappear from the predicate which has now been linked with its argument.

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 (51) comes from Italian.

(51) Adoro Luisa.

admire.1sg Luisa

‘I admire Luisa.’

The inflectional phi-features are extracted from the input and embedded as morphosyntactic features to the corresponding lexical items, as shown in (52).

(52) John admire + s Mary.

↓ ↓ ↓

[John [admire Mary]]

[…3SG…]

This means that *admires* will have both unvalued (from the lexicon) and valued phi-features (from the input) 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 (53).

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

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

c. = [φ_] + [φ].

If the predicate head has valued phi-features, then AgreeLF must also check that there is no phi-feature conflict. A sentence such as **Mary admire John* will be recognized as ungrammatical. AgreeLF is implemented by internal search (§4.8.8).

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 the 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 (an arbitrary) symbol that can (but need not) 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 an ordinary text file the user can modify.

Let us consider the processing of morphologically complex words. The lexicon matches phonological words with a *morphological decomposition*, if the latter is present. A morphological decomposition is a linear string of morphemes m_1#...#m_n. These morphemes, which (we assume) designate primitive lexical items, retrieve the corresponding primitive lexical items $M_1 + \ldots + M_n$ from the same lexicon (i.e. M_1,...,M_n must be keys in the same lexicon). These items are fed into syntax, where they form complex heads $M_1(M_2\ldots(M_n))^0$. We can imagine the morphologically complex words as "morphological chunks," lexical information that has been packaged into one unit. Cyclic and noncyclic head reconstruction (§ 4.8.6) extracts them into head spreads $[M_1\ldots[M_2\ldots[\ldots M_n\ldots]]]$ (54).

(54) a. John admires Mary. (Input)

b. John T+V Mary (Output of lexicon, morphological decomposition)

c. John $T(V)^0$ Mary (PF, input to syntax)

d. [John [T [V Mary]]] (LF, output of head reconstruction)

Primitive lexical items are feature sets $\{f_1..f_n\}$(with one additional detail, discussed below) . 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 ~ $\{f_1...f_n\}$).

Let us consider an example, processing of sentence *John admires Bill*. Each phonological word is processed as a separate item:

```
Phonological words: ['John', 'admires', 'Bill']
    Sensory Processing of /#John/
    Sensory Processing of /#admires/
    Sensory Processing of /#Bill/
```

The processing then begins with the first phonological word /John/

```
Next morpheme /John/ retrieves  (1) morphological chunk [JOHN.D]
Next morpheme /.D_/ retrieves  (1) D
```

which retrieves a morphological chunk `[JOHN.D]` containing two items JOHN and D. If the item is ambiguous, i.e. there are several lexical entries matching with the key /John/, they are all listed

here. Both JOHN and D items are keys in the same lexicon: JOHN refers to the N-head storing lexical content, D refers to the lexical entry for the default D-head. The two items are separate by comma, because they are packed inside the same morpheme [John]. The two items /John/ and JOHN are specified in the following lexical entries

```
823  John        :: "JOHN.D" LANG:EN
824  JOHN        :: PF:John LF:John N R_exp sg 3p hum m def -SEM:directional nonadjoinable LANG:EN
```

where the items following JOHN specify its lexical features, including its lexical category N. Both JOHN and D are mapped with the respective primitive lexical items JOHN^0 and D^0:

```
16      Next morpheme /.D_/ retrieves  (1) D
17
18      Next head D°
19
20      Next morpheme /_JOHN./ retrieves  (1) JOHN
21
22      Next head JOHN°
```

Symbols JOHN^0 (N^0) and D^0 indicate that we have moved from the domain of lexical entries and lexical items into primitive constituents or heads. For example, the list of features that appear on line 824 in the lexical entry of JOHN are now included inside N^0. These two heads are then packed inside one complex head at the PF

```
26  ⊢   Merge(D, John)
27      = D(John)
```

and extracted into [$_{DP}$ D N] by cyclic head reconstruction (IHM):

```
29      Cyclic reconstruction:
30
31      Feature inheritance(D(John))
32       = D(John)
33      IHM(John)
34       = [D John]
```

The sequence is therefore: /John/ ~ JOHN.D ~ D(JOHN)^0 ~ [$_{DP}$ D N]. The reason for the existence of complex head D(JOHN)^0 at PF is nonlocal head movement; a model in which this step is skipped and JOHN.D is mapped directly in [$_{DP}$ D N] gets us quite far does not capture everything. Phonological strings mapping into primitive lexical items (lines 333-338 below) and strings

mapping into complex decompositions (line 332) 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.[48]

```
332  maalat#ti#in     :: "MAALAA.asp#impass.i#Vn" LANG:FI
333  maalaa/2         :: "MAALAA.v.0#[-V]" LANG:FI
334  'paint.prs.3sg'
335  maalaa           :: "MAALAA.v.0" LANG:FI
336  'paint.prs'
337  MAALAA           :: PF:maalaa- LF:paint V V/TR LANG:FI
338  'paint'
```

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. An 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. 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*).[49]

Consider again the following screenshot from the file defining the content lexicon, specifically, the some of the entries for the verb ‘paint’.

---

[48] It is possible to feed the algorithm with any string that exist in the lexicon. D ⊕ JOHN ⊕ *T* ⊕ *v* ⊕ *admire* ⊕ D ⊕ MARY can generate the same outcome as *John admires Mary*, the only difference begin that steps (54)a–c are omitted. While this is an unrealistic sensory input, it can be used to model language production.

[49] We can bypass the 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’.

```
322  maalat#essa     :: "MAALAA.v#ESSA/inf" LANG:FI
323  maalat#tua      :: "MAALAA.v#TUA/inf" LANG:FI
324  maalaa#van      :: "MAALAA.v#VA/inf" LANG:FI
325  maalat#en       :: "MAALAA.v#E/inf" LANG:FI
326  maalaa#massa    :: "MAALAA.v#MAine/inf" LANG:FI
327  maala#ta        :: "MAALAA.v#A/inf" LANG:FI
328  maala#si        :: "MAALAA.v#i+0" LANG:FI
329  maalas#i#n      :: "MAALAA.v#i#n" LANG:FI
330  maalan#nut      :: "MAALAA.v.asp#nUt" LANG:FI
331  maala#ta#an     :: "MAALAA.asp#impass.0#Vn" LANG:FI
332  maalat#ti#in    :: "MAALAA.asp#impass.i#Vn" LANG:FI
333  maalaa/2        :: "MAALAA.v.0#[-V]" LANG:FI
334  'paint.prs.3sg'
335  maalaa          :: "MAALAA.v.0" LANG:FI
336  'paint.prs'
337  MAALAA          :: PF:maalaa- LF:paint V V/TR LANG:FI
338  'paint'
```

Items 322–332 represent complex phonological words (keys in the lexicon dictionary) which map into morphological decompositions, strings of further entries in the same lexicon. On the left are the phonological words or keys which are used to retrieve the words and which must exist in the input. The morpheme boundaries marked by # are optional, and will be removed during the retrieval phase, so they should not appear in the input: their purpose is to illustrate how the phonological word is assumed to break down into its analysis appearing the right (that is, this information helps to understand the structure of the words). The analysis, or gloss, is then on the right: period "." corresponds to fused (portmanteau) morphemes, # to a regular morpheme boundary, = to clitic boundary and | to prosodic feature (the + on line 328 is part of the morpheme symbol and does not indicate a boundary type). Different boundary types are interpreted slightly differently. Each analysis begins with the root stem MAALAA whose properties are provided as a feature set on the last line 337.[50] This item is expressed (for convenience) by capital letters. All morpheme symbols appearing in the analyses must be found from the lexicon, so that they can be mapped into appropriate feature sets during lexical retrieval. For example, there is a separate lexical entry for VA/inf, designating and heading a particular infinitival clause construction in Finnish. As pointed out above, the symbols are arbitrary: in this particular instance they try to model a linguistically correct and realistic analysis of the target words. But the downside is that

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

in order to find out what the different symbols mean, the user must consult the same lexicon.[51] In short, the symbols used to recognize and retrieve the lexical items can be anything; the lexical features must correspond to something the model can interpret.

It is possible to have the same phonological string to map into two or more lexical entries. This creates ambiguous words. When the model confronts an ambiguous word, all its lexical entries create their own parsing subtrees. This can be avoided if not required for research purposes by disambiguating the word manually in the input string. In the above screenshot from the Finnish lexicon, there are two entries `paint` and `paint/2` which serve this purpose.

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 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, already shown in the previous section, 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

---

[51] Consider, for instance, the fact that the Finnish finite present tense is expressed by a phonologically null morpheme symbolized by `0` in the above entries: the reader might not know that `0` represents tense unless the corresponding entry is first located from the lexicon. An alternative is to use a symbol such as `T/prs`. It is, of course, possible to list both entries in the lexicon and use either symbol, or some other symbol.

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. 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. Finally, 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 are also 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). 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 adpositions as an example. Adpositions are represented by the 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 exists a special group of adpositions which take finite clause complements, it is possible to override the lexical redundancy rule by providing the required selectional features in the language-specific lexicon.

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 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 take the postpositional form (*lähellä minua* 'near I.PAR', *minun lähellä-ni* 'I.gen near-PX/1SG'). This 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 mapped into primitive lexical items and streamed into syntax where they form complex heads and then head spreads via head reconstruction (§4.8.6). Inflectional features are mapped into feature sets which are inserted inside an adjacent lexical item in word and do not create their own primitive heads. 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 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. There is some evidence that agreement can project separate heads rather than just features, sometimes the issue is unclear (as in the case of Finnish possessive suffixes). Here is what happens during the processing of /admires/:

```
35      Next morpheme /admires/ retrieves  (1) morphological chunk [admire-#v#T/fin#prs#[-s]]
36      Next morpheme /#[-s]_/ retrieves  (1) [-s] = [LANG:EN][PER][PHI:NUM:SG,PER:3][inflectional][ΦPF]
37      Next morpheme /#prs#/ retrieves  (1) prs = [LANG:EN][inflectional][prs]
38      Next morpheme /#T/fin#/ retrieves  (1) T/fin
```

First, the item is decomposed into its constituent morphemes (line 35). /-s/ and /prs/ are mapped into inflectional feature sets (lines 36, 37). Because they are inflectional feature sets, they do not project separate grammatical heads; rather, the structure will be composed from T, v and V:

```
91        = [[D John]:2 [T [__:2 [v admire]]]]
```

such that T contains the feature retrieved on the basis of [prs] and /-s/ (lines 191, 192):

```
188          T          {Fin T π Φ EF* PF:T LF:T +SELF:ΦLF PHI:DET:_ PHI:NUM:SG ΦLF PHI:IDX:§1 PHI:PER:3
189                      +SELF:ΦLF +COMP:ASP,T/prt,COPULA +SELF:ΦLF LANG:EN !PER IDX:1,PRE T/fin
190                     TAM§3}
191                     {LANG:EN prs}
192                     {PHI:NUM:SG,PER:3 ΦPF LANG:EN PER}
```

The above screenshot shows the feature content of T at the LF-interface. The reason T is composed of three separate feature sets is explained in §4.9.4.

4.9.4 Lexical items and their core

While the assumption that primitive lexical items consist of feature sets provides a useful starting point, the system seems to be slightly more complex due to the existence of clitics. Another reason for the complications described in this section is that we need a mechanism that regulates accepted inflectional affix sequences. The system elucidated above allows the system to process, interpret and accept input strings such as *talo#n#en#en…* 'house-gen-gen-gen…' with an unbounded number of the same affix that would all map into the same feature. Furthermore, since the inflectional affix sequences are mapped into unordered sets, their order in the input words become irrelevant. Finally, in languages like Finnish the same nominal element may possess several conflicting phi-features: one set from concord, another from nominal (possessive) agreement. All of these facts point to the conclusion that there is more to the inflectional system than just a set of features.

Before describing the solution to these problems, let us consider briefly the idea that we regulate the affix sequences by relying on an independent presyntactic morphological component. A presyntactic morphological component would check or filter morpheme sequences inside the lexico-morphological system. The assumption that the system operates inside the lexico-morphological component faces the difficulty that at that stage of processing we do not have full lexical items at hand and can operate solely with the phonological properties of the morphemes,

which is insufficient for most purposes.[52] Recall that since the lexical items are retrieved and instantly feed into the syntactic pipeline, there is no representation in which the lexical items form "linear strings." The assumption that lexical items are arranged linearly at some processing stage would require major architectural changes to the current LPG. An additional possibility is to add the morphological checks to the syntactic-morphology interface which feeds syntax with lexical items. This would, however, require that we complicate the interface by positing a memory system able to verify the correctness of lexical item sequences. A more plausible approach would be to make illicit morpheme sequences unrecognizable by requiring that they satisfy conditions on morphological contexts. This approach seems plausible, however it is unlikely to provide a complete solution since in some cases the principles seem to be more syntactic or semantic in nature.[53] For these and other reasons it is assumed that the morphological ordering principles are capture syntactically or semantically; the first linguistic item able to consider linearly ordered lexical items is the PF. Still, inflectional features by current assumptions constitute sets, not linearly ordered sequences.[54] For the these and other reasons it is assumed that the order of inflectional affixes is preserved throughout the processing, but the problem is that this is not compatible with the assumption that the features form unordered sets.

Let us consider in some more detail the problem of independent agreement clusters such as clitics and nominal agreement. Romance pronominal clitics constitute independent pronominal agreement clusters that are attached to hosting elements such as auxiliary or regular verbs and form phonological words together with their hosts. They do not represent agreement, but independent pronouns. A somewhat analogous situation obtains in the case of Finnish nominal agreement clusters, which can be combined with independent concordial agreement. Thus, a word

---

[52] Excluding purely morphophonological regularities that clearly belong to the presyntactic component and which has not yet been modelled by using LPG.
[53] This hypothesis is not part of the current LPG since developing and testing it requires a special dataset and study focusing on morphology.
[54] Derivational morphemes can already be specified to be recognized only within specific morphological contexts, which models the bound/free morpheme distinction.

such as *auto-ja-ni* 'car-PAR.PL-PX/1SG' meaning 'my cars' contains two conflicting agreement features, the concordial plural marking and the singular first person possessive suffix corresponding to a separate possessive argument 'my' that may remain covert. In both cases inserting the independent agreement clusters, the pronominal clitics or the possessive agreement suffixes, directly inside their hosts will introduce phantom phi-feature conflicts. The two phi-feature clusters are treated as independent packages of information.

These observations and several others led to a slightly more complex model in which the lexical item does not consists of a single set of lexical features but a linear sequence of lexical feature sets (called *feature bundles*) <{L…}, …>. The feature bundles (separate sets) are created from inflectional affixes separated by regular morpheme (#) or clitic (=) boundaries in the input words, while the features of fused portmanteau morphemes are inserted inside the same bundle. The result is that the concordial phi-features are represented inside a separate bundle than the possessive features, and the two do not mix with each other. It is assumed, furthermore, that Romance pronominal clitics are independent pronominal phi-feature bundles inside their hosting verbs.[55]

4.9.5 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 derivations containing [F] and [−F].

[PF:X] provides the phonological matrix of the element, currently just a string. [LF:X] provides its idiosyncratic semantic interpretation which is supposed to function as a link into conceptual

---

[55] The system is implemented by encapsulating the feature bundling system inside its own module called the *core* of the lexical item. In this way we can change its internal structure and operations without the need to implement corresponding changes anywhere else in the code. For example, the core has a method which returns all features as single set and thus ignores the internal bundling.

representations and encyclopedic knowledge (this will be replaced later by a lexical copy-link). [+COMP:L,…] and [–COMP:L,…] mandate or block complements of the type L…, where L… is a list of features inside the head of the complement but usually refers to one of the major lexical categories. The relevant notion of complement is defined by external search paths (§4.6). The head is determined by labeling algorithm §4.5. [+SPEC:L…], [–SPEC:L…] function in the same way. [TAIL:L…] requires that the element tails a head with feature(s) L….

4.10 Narrow semantics

4.10.1 Syntax, semantics and cognition

Semantics is the study of meaning. 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, events, thoughts or propositions as represented by the hearer and the speaker. The temporary semantic repository is called *global discourse inventory*. The discourse inventory can be accessed by global cognition, such as thinking, decision making, planning, problem solving and others. Linguistic expressions are utilized to introduce, remove and update entities in the discourse inventory. We can think of linguistic conversation as 'updating' the global discourse inventory.

The global discourse inventory is an internal system defined by semantic representations in the minds of the speaker and addressee; the corresponding real-world objects, if there are any, are not modelled in this study. If the speaker uses a proper name such as *John*, a corresponding semantic object is projected into the global discourse inventory of the addressee and speaker irrespective of whether any such person exists in the real world. It could be a fictious person that only the speaker recognizes as such, or a fictious object, such as *the present king of France*, that both interlocutors accept as such. This internal reality is constructed as the conversation proceeds completely irrespective of what happens in the external world, and therefore also the truth conditions or the veridicality of the sentences appearing in the conversation are represented nowhere. Since grammar does not seem to make a robust distinction between fiction and reality,

we proceed on the assumption that the latter notions belong to some other cognitive component, or perhaps it is a matter of interaction between mind and the external reality. Thus, the semantic systems as conceived here do not provide sentences with truth conditions.

The hypothesis that linguistic expressions provide a vehicle for updating the contents of the discourse inventory requires that there exists some mechanism which translates linguistic signals into changes inside the global discourse inventory. This mechanism, or rather collection of mechanisms, is called *narrow semantics*. The term "narrow" is meant to distinguish this system from "broad semantics" which cover semantics as a whole. 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 information 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. For example, information structure (notions such as topic and focus) is created by different processing pathway than the system that interprets quantifier scope. Narrow semantics can therefore be also viewed as a gateway 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. It is specifically "narrow" in the precise sense of mediating communication between language and cognition.

Referential expressions create special problems due to ambiguity. 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*). Even *John* can be ambiguous if there are several men with the same name or several unnamed male persons. The general problem is that there is nothing in the expression itself that determines unambiguously what it denotes, so the listener must perform

disambiguation. Thus, all expressions are first linked to unambiguous semantic entries inside narrow semantics, representing their intrinsic semantic properties – intensions – and these intermediate representations are transformed into actual denotations – extensions – 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 makes a claim about some "it" that we must link with something. This pronoun can denote four things in this particle context: the horse, the barn, the whole event, or a third entity not yet mentioned. 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). When all referential expressions in any given expression are provided with a denotation (from the set of possible denotations), we call the resulting mapping an *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, however, provides all possible assignments and their rankings as output. Assignments are calculated at the language-cognition interface. When referential expressions have several possible denotations, the hosting expression will have several assignments.

When a linguistic feature such as [SG] 'singular' is transformed into a format understood by global cognition, what we mean is that the formal signal representing that feature in the linguistic output (and the corresponding lexical feature inside some lexical item's core, § 4.9.4) 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.

The above discussion considered cases where referential expressions such as *John* project individual entities into the global discourse inventory, which can then be referenced later on. Referential expressions are not, however, the only type of expressions that can project entities into the global discourse inventory. Clausal structure projects events and propositions, while lexical root stems project concepts; all these and whatever other objects are posited by the theory populate the global discourse inventory. The projections are triggered by specialized lexical features. For example, the ability for tense-aspect heads to project and denote events is determined by a special lexical feature that triggers this behavior inside narrow semantics. Events and propositions are furthermore related to their participants; in general the representations projected in the global discourse inventory should be in a format that global cognitive processes can understand, whatever that format might be (currently they are Python dictionaries).

Let us illustrate all of the above by considering the processing of a simple sentence *John admires Bill*. The LF-interface representation which feeds semantic interpretation is

```
142
143       = LF-interface [[D John]:2 [T [__:2 [v [admire [D Bill]]]]]]
144
```

where both DPs *John* and *Bill* are interpreted as referential expressions denoting 'thing' objects. This interpretation is due to the fact that both DPs are headed by D-head containing a set of features allowing them to function in this way. In the current implementation (version 20.1) the following semantic objects are projected into existence on the basis of the above LF:

```
148     Object projections:
149         Project object (1, QND) for [D John]
150         Project object (1, GLOBAL) for [D John]
151         Project object (2, QND) for [D Bill]
152         Project object (2, GLOBAL) for [D Bill]
153         Project T-event (1, PRE)
154         Project object (3, GLOBAL) for T-Event
155         Project N-concept (2, PRE)
156         Project object (4, GLOBAL) for Concept John
157         Project V-concept (3, PRE)
158         Project object (5, GLOBAL) for Concept admire
159         Project N-concept (4, PRE)
160         Project object (6, GLOBAL) for Concept Bill
```

Objects with the label `GLOBAL` are projected into the global discourse inventory. `QND` refers to "quantifiers, numbers and determiners" and contains narrow semantic representations (intensions) for referential expressions; `PRE` refers to "properties, relations and events" and contains narrow semantic representations (intensions) for one event (T-event 'John admires Bill') and three concepts, the concepts of admiring, being John and being Bill. The same objects also exist in the global discourse inventory. Expressions such as *nobody* would have narrow semantic intensions but no extension in the global discourse inventory. The possible dentations for the two referential expressions *John* and *Bill* are calculated next as follows:

```
164     Denotations:
165         [D John]~['1', '2']
166         [D Bill]~['1', '2']
```

Numbers [1] and [2] refer to the object identifiers in the global discourse inventory, as listed above (lines 150, 152). Thus, *John* can refer to John or Bill, and *Bill* can refer to John or Bill. Both names can refer to both objects, since there is no requirement in language which says that a name such as *John* could not denote a person that is elsewhere denoted by a different name. Perhaps *John* is Bill's nickname, Bill is commonly referred to by using two (or more) names, or the speaker is confusing the addressee by a mistake or by purpose – names are arbitrary symbols for things.[56] The fact that two *separate* persons must be at stake is determined at the level of assignments and not ruled out when considering possible denotations:

---

[56] We could of course rule out mappings like *John* ~ Bill by positing a semantic feature 'has a name N' at the level of global discourse inventory and then force a condition which uses names

```
167     Assignments:
168         Assignment [D John] ~ 1, [D Bill] ~ 1 -
169         Assignment [D John] ~ 1, [D Bill] ~ 2 +
170         Assignment [D John] ~ 2, [D Bill] ~ 1 +
171         Assignment [D John] ~ 2, [D Bill] ~ 2 -
172         Summary:  John[a] admire Bill[b]
```

+ indicates that the assignment was accepted, – that it was rejected (due to the binding condition C). The two accepted assignments are: one where *John* refers to Bill and *Bill* refers to John, and another where *John* refers to John and *Bill* refers to Bill. This information is summarized in the last line (172).

4.10.2 Argument structure

The notion of argument structure refers to the way referential arguments are organized syntactically and semantically around their predicates referring to events, properties and relations. Consider the situation depicted in the following figure.

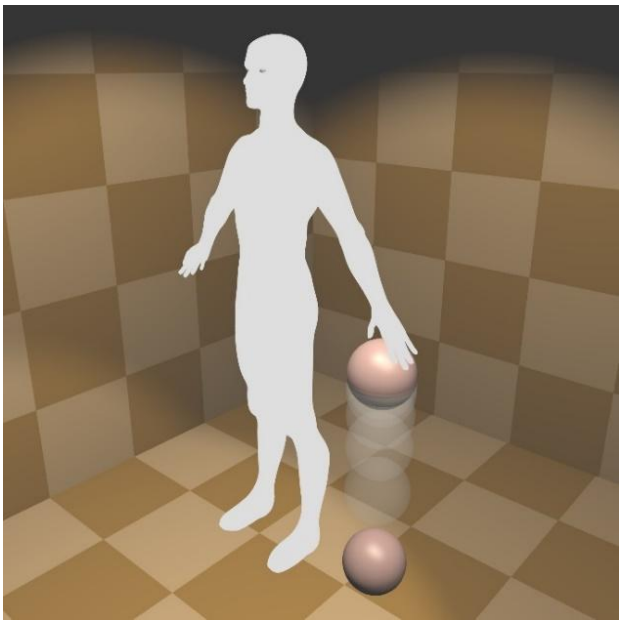


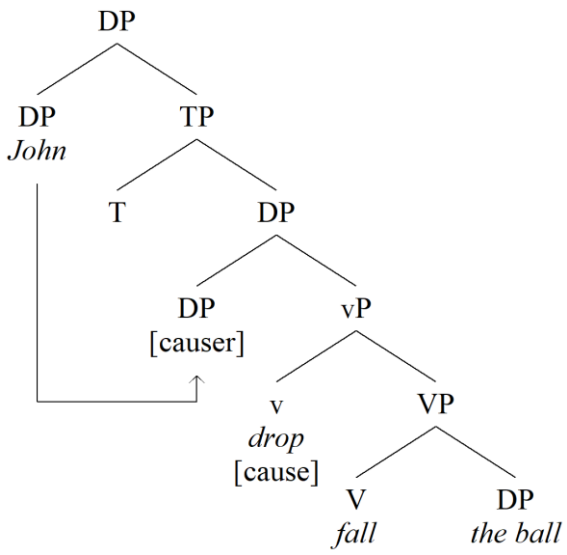
Fig. A non-discursive, perceptual or imagined representation of the meaning of the sentence *John dropped the ball*.

---

like phi-features by restricting the set of possible denotations. But no such limitation is in operation in language: names are arbitrary symbols for thing objects.

Imagine we experience a non-discursive (analogous, continuous) event of a person dropping a ball: 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*, on the other hand, grammaticalize these 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 *John dopped the ball* and how the different parts may correlate with the conceptual experience depicted in the above figure.

(55)



The combination of the object *the ball* and the verb – the lower VP – corresponds 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. John is the *agent* of the event

because he literally causes the ball to fall. The combination of *fall* + v will be the transitive verb *drop* 'cause to fall', something we call a *complex predicate*. It is complex because it is composed of two parts. 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 and causes the system to project a T-event in the global discourse inventory ('there is/was an event in which…'), but the argument at SpecTP will not get a secondary thematic interpretation. It is assumed that tense is a way of denoting the event itself, and consequently projecting it into the global discourse inventory. The three-part predicate T + v + V forms a complex predicate. Here is the semantic object corresponding to the T-event in the global discourse inventory:

```
243         Object T-Event(3) ['§T-Event'] in GLOBAL
244             Composition: T + DP + v + V
245             Participants: [None, '2', '1']
246             Predicate: T
247             Reference: T-Event
248             Semantic space: GLOBAL
249             Semantic type: ['§T-Event']
```

Not all transitive predicates project causation into the event they denote. Sentence *John resembles Mary*, for example, does not. Instead, causation is a special case of temporal precedence: the agent argument is perceived to participate in the event before the patient, and causation is only one way of how this may come into being. Because constituency structure mirrors linear order, it could be that the temporal ordering of the arguments at the LF-interface mirrors the temporal ordering of the participants in the event, and this is ultimately how LF-interface representations are interpreted. Thus, in the above example of *John dropped the ball*, John is interpreted as the causer because his involvement in the event precedes the ball's involvement. John initiates the event.

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 a 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 semantic requirements grounded in the structure of human experience.

4.10.3 Predicates, arguments and antecedents

Predicates are characterized by the fact that they must be linked with arguments (§4.8.9). A predicate is defined in the lexicon as such by unvalued phi-features [φ_], representing an "unsaturated argument" in the Fregean sense (this is represented by symbol Φ which creates all unvalued phi-features to the predicates depending on language via lexical redundancy rules). A predicate α is linked to an argument at the LF-interface/narrow semantics by finding the closest possible argument in its vicinity, thus from α's (i) own feature content (sublexical pro-elements), (ii) complement, (iii) specifier(s) and finally (iv) any constituent within the external search path (21) §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.
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.

Failure to find an argument from αP results in a situation where the argument is searched from a nonlocal domain by external search (56)d–e, capturing control.

The argument-predicate mechanism interacts with AgreeLF. When the argument is located *inside* the complement, it is invisible for the predicate-argument mapping rule. AgreeLF 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 §4.8.5.[57] Because argument identification makes use of AgreeLF, it will also rely on internal search path.

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

[57] 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).

('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 (or Ā-operators)

Operators are elements that create operator-variable dependencies, usually (perhaps exclusively) represented by Ā-chains §4.8.2, 4.8.4. Interrogative clauses (57) represent a prototypical example of operators and operator-variable structures.

(57) What did John found __?

'which x: John found x'

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 search path with [wh][FIN]. The finite operator cluster [WH][FIN] is created during reconstruction, as explained earlier §4.8.2, 4.8.4. The relevant configuration is illustrated by (58).

(58) What did$_{wh,fin}$ 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'.

One special feature of the operator-variable module is that in order to create a coherent interpretation, the system must detect two constituents: the operator itself, and the scope marker inside the external search path. Thus, the interpretation relies on discontinuous constituents or linguistic objects. The same is true of predicate-argument pairs. It is not an accident that both operators and arguments undergo copying, which creates the corresponding syntactic discontinuity inside syntax and ultimately in the surface sentences.

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 (§4.10.1). 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. In the same way, *she* cannot denote a male person.

Some referential expressions 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 filters possible assignments. For example, these filtering mechanisms block all assignments in which *John* and *Bill* in *John admires Bill* refer to the same person:

```
Assignments:
    Assignment [D John] ~ 1, [D Bill] ~ 1 -
    Assignment [D John] ~ 1, [D Bill] ~ 2 +
    Assignment [D John] ~ 2, [D Bill] ~ 1 +
    Assignment [D John] ~ 2, [D Bill] ~ 2 -
    Summary:  John[a] admire Bill[b]
```

Rejected assignments are marked by –, accepted by +.

The binding filters operate at the language-cognition interface by blocking illicit assignments. The mechanism relies on the lexicalist binding features, the notion of syntactic working memory, itself based on external search (§4.6), and intervention. The binding features that are part of referential expressions (regular pronouns, reflexives and r-expressions) restrict assignments by demanding that the denotation for expression, say E, must be either new or old in relation to

denotations of other referential expressions in the semantic working memory, where semantic working memory contains semantic objects targeted by external search from E. If E must be new, as specified by its lexical binding feature, then all coreference dependencies between E and other referential expressions reached by external search from E are blocked, corresponding to the binding profile of r-expressions (62).

(62) a. *John$_a$ admires John$_a$.

b. *John$_a$ believes that Mary admires John$_a$.

c. John's$_a$ brother admires John$_a$.

In (62)c the embedded *John* is outside of the external search domain from the lower *John*, hence a coreference relation is possible (but not, of course, necessary). Coreference dependencies (62)a–b are ruled out at the level of assignment generation, thus there are no syntactic devices inside narrow syntax such as indices which would block them. We get the binding profile of regular pronouns (63) if we add a locality restriction to the mechanism.

(63) a. *John$_a$ admires him$_a$.

b. John$_a$ believes that Mary admires him$_a$.

c. John's$_a$ brother admires him$_a$.

The coreference relation (63)a is blocked because the denotation of *him* must be new in relation to what is contained inside local semantic working memory, where locality is defined by relying on intervention, more specifically intervention by another referential expression as shown in (63)b. Nonlocal coreference dependencies (63)a as well as coreference dependencies outside of the working memory (63)c are possible. Reflexives are pronouns whose denotation must be old inside the same locality domain (64):

(64) a. John$_a$ admires himself$_a$.

b. *John$_a$ believes Mary admires himself$_a$.

c. *John's$_a$ brother admires himself$_a$.

Removing the locality restriction yields long-distance reflexives. In this way, the binding features are like indefinite and definite determines which restrict the search space for assignments (definite determiner denotes an old, indefinite a new object), but they rely on additional structure-sensitive external search on LF and in so doing further limit the domain of candidate denotations when the expression is disambiguated at the language-cognition interface. The results are reported in the derivational log file and, in a more condensed form, in the results file. The information is in three segments in the derivational log file. First, there is a summary of the objects projected into the global discourse inventory from which all denotations and assignments are generated (lines 148-160); this is followed by a list of possible denotations for each referential expression (164-166); and finally all accepted and rejected assignments are shown (lines 167-172). The last line 172 contains a summary.

```
148     Object projections:
149         Project object (1, QND) for [D John]
150         Project object (1, GLOBAL) for [D John]
151         Project object (2, QND) for [D Bill]
152         Project object (2, GLOBAL) for [D Bill]
153         Project T-event (1, PRE)
154         Project object (3, GLOBAL) for T-Event
155         Project N-concept (2, PRE)
156         Project object (4, GLOBAL) for Concept John
157         Project V-concept (3, PRE)
158         Project object (5, GLOBAL) for Concept admire
159         Project N-concept (4, PRE)
160         Project object (6, GLOBAL) for Concept Bill
161     Argument for T°: [D John], indexed to [D John]
162     Argument for v°: [D Bill], indexed to [D Bill]
163     Argument for admire°: [D Bill], indexed to [D Bill]
164     Denotations:
165         [D John]~['1', '2']
166         [D Bill]~['1', '2']
167     Assignments:
168         Assignment [D John] ~ 1, [D Bill] ~ 1 -Illegitimate binder for [D Bill](1) in WM
169         Assignment [D John] ~ 1, [D Bill] ~ 2 +
170         Assignment [D John] ~ 2, [D Bill] ~ 1 +
171         Assignment [D John] ~ 2, [D Bill] ~ 2 -Illegitimate binder for [D Bill](2) in WM
172         Summary:  John[a] admire Bill[b]
```

It is possible to write correct, native-speaker assignments into the dataset by using the summary notation on line 172 (see the screenshot below), which causes the program to compare automatically the predicted output with the gold standard output and report all mismatches.

```
# Condition C (1-3)

    John admires Bill
    ¦-> Binding: John[a] admire Bill[b]

    he `s brother admires Bill
    ¦-> Binding: he[b] brother[a] admire Bill[b,c]

    Bill said that John admires Tim
    ¦-> Binding: Bill[a] say John[b] admire Tim[c]

# Condition A (4-6)

    John admires himself
    ¦-> Binding: John[a] admire self[a]

    he `s brother admires himself
    ¦-> Binding: he[b] brother[a] admire self[a]

    Bill said that John admires himself
    ¦-> Binding: Bill[a] say John[b] admire self[b]

# Condition B (7-9)

    John admires him
    ¦-> Binding: John[a] admire he[b]

    he `s brother admires him
    ¦-> Binding: he[b] brother[a] admire he[b,c]

    Bill said that John admires him
    ¦-> Binding: Bill[a] say John[b] admire he[a,c]
```

5 Performance

5.1 Introduction

The linear phase theory (LPG) is a model of human language comprehension and production. 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. I will conclude by presenting a few words concerning language production.

It is perhaps useful to repeat some of the points made earlier in §3.1 concerning the relation between competence and performance. Any empirically interesting LPG model must be observationally, descriptively and (ideally) explanatorily adequate, and there must exist a rigorous, unambiguous demonstration that this is so. Aspects of performance – if considered at all – should not be added at the expense of linguistic realism. That being said, LPG was designed from the beginning as a model of grammar that could take performance facts into consideration. We can depict it as an extension of the standard bottom-up model such that the derivation is created in a psycholinguistically realistic way. That is, we embed the insights accomplished with the help of the standard bottom-up models into a left-to-right cycle and then simply add performance properties to the latter.

5.2 Mapping between the algorithm and brain

The figure below 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, 2023a).

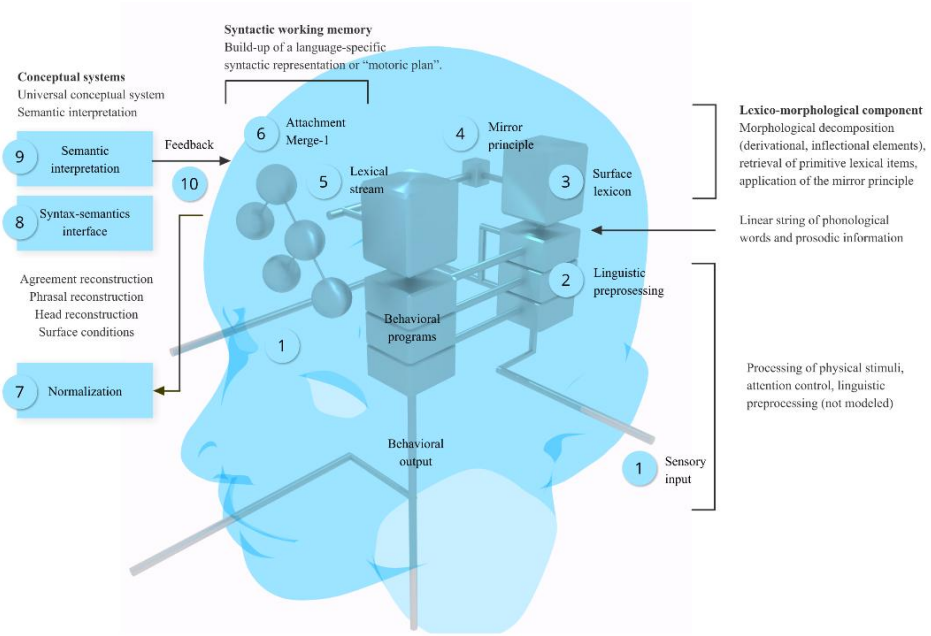


Fig. Components of the model and their approximate locations in the human brain. See the main text for explanation.

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 cyclic and/or noncyclic transfer (§4.8), as damage to the Broca's region affects 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.

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 at a later stage. Some transformational repair operations are performed cyclically when the input words are consumed.

There is evidence that the human language comprehension is incremental. It is possible to trick the system into making a wrong decision on the basis of incomplete local information. This can be seen from (65), 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.

(65)

a. The horse raced past the barn.

b. The horse raced past the barn fell.

The linear phase algorithm interprets *raced* locally as a finite verb and ends up in a dead end when processing *fell*, backtracks, and consumes additional cognitive resources before it finds the correct analysis. 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.

We do not assume, however, that exhaustive backtracking is 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 explore the complete parsing tree. If we only generated the first solution, spurious secondary solutions might escape attention. It is not at all 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 spent in processing different types of constructions and/or different algorithmic solutions. In addition, realistic language comprehension 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.[58]

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.

---

[58] It is possible and 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, but this remains a conjecture and has not been implemented, let alone tested.

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 disconnected phrase structures. 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 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 a 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 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 local attachment solutions to be 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 (66) is first attached to the lowest possible solution, and only if that solution fails, to the nonlocal node.

(66) 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. 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, 6.2.3). 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. 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.

5.3.7 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. 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 (§6.3.5) 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 language production

Language comprehension maps linguistic sensory inputs (linear sequences of phonological words) into PF-interface representations (SM-PF mapping), LF-interface representations (PF-LF mapping) and finally into semantic interpretations (LF-SEM mapping). The first step is specific to comprehension, since it deals with phonological words, ultimately with concrete sensory inputs. Production is modelled by generating PF-interface representations from a linear sequence of zero-level phrase structure objects (simple and complex heads) and mapping them into SM and LF on a phase-by-phase basis, when left branches constitute phases. Thus, language production, like comprehension, proceeds cyclically in a left-to-right order, and the output is generated in tandem with the derivation.

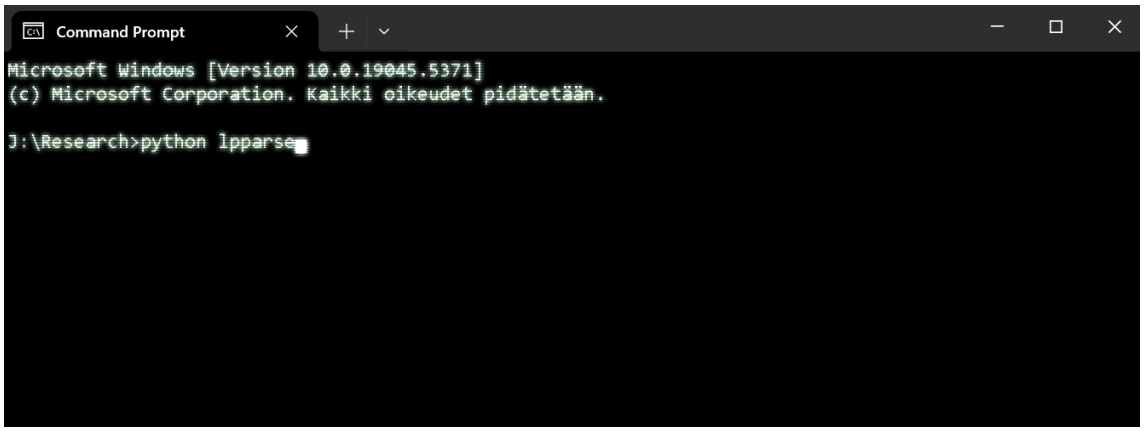
6 Inputs and outputs

6.1 Installation and use

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

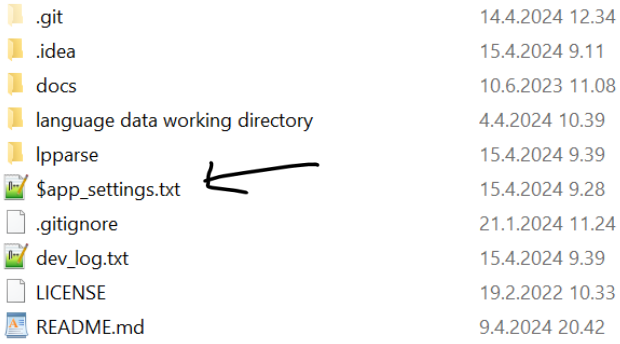
`https://github.com/pajubrat/parser-grammar`

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 on the local computer. Refer to the Python installation guide (`www.python.org`). 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. This information is defined in the following way. The root directory contains a file `$app_settings.txt`.



This file contains a pointer to a `study configuration file` (*.lpg). For example:

```
1  file_study_configuration=binding_theory.lpg
2  file_study_folder=language data working directory/study-10-binding-theory
```

In this case we use the study configuration file `binding_theory.lpg` (see Section 6.2.1) that is located in the `study folder` provided in the second line.

6.2 Structure of the input files

6.2.1 Study configuration file

6.2.1.1 General

Each study is defined by its own *study configuration file* that sets the parameters in a way intended by the researcher. The study configuration file is a list of parameters, each containing a key and a value separated by =. For example, the study configuration file `binding_theory.lpg` begins with the following key, value pairs:

```
author= Pauli Brattico
year= 2025
date= February
study_id= 10
study_folder=           language data working directory/study-10-binding-
test_corpus=            binding_theory_corpus.txt
numeration=             X

# GENERAL SIMULATION PARAMETERS
general_parameter_only_first_solution = True
ignore_ungrammatical_sentences = False
console_output = Full
stop_at_unknown_lexical_item = True
logging = True
use_numeration = False
dev_logging = True

# FILES
file_study_folder=language data working directory/study-10-binding-theory
file_test_corpus=binding_theory_corpus.txt
file_lexicon_folder=language data working directory/lexicons
file_lexicons=lexicon.txt;UGmorphemes.txt;inflections.txt
file_redundancy_rules=redundancy_rules.txt
```

The key determines the name of the parameter. For example, the key `file_study_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. The parameters are the following. The files section defines the names and locations of the external input files used in the simulation.

6.2.1.2 General simulation parameters

```
# GENERAL SIMULATION PARAMETERS
general_parameter_only_first_solution = True
ignore_ungrammatical_sentences = False
console_output = Full
stop_at_unknown_lexical_item = True
logging = True
use_numeration = False
dev_logging = True
```

Simulation parameters can be either True or False. They have the following interpretations: `only_first_solution` will stop the derivation when the first legitimate output analysis has been found. This speeds up processing when examining observational adequacy, but the model fails to find ambiguities. `logging` will set logging on or off. `use_numeration` will generate the input dataset from a numeration (not used in any actual study so far). `assignments` will attempt to calculate assignments and is involved in binding. `pragmatics` calculates information structure and other pragmatical attributes. `thematic roles` calculates thematic roles.

`project_objects` will project semantic objects into the global discourse inventory (ontology). `predicates` will calculate predicate-argument dependencies.

6.2.1.3 Image parameters

Image parameters determine how phrase structure images are generated. Currently these images can only be generated by using the graphical user interface (Chapter 9).

```
# IMAGE PARAMETERS

# General
image_parameter_words = True
image_parameter_glosses = True
image_parameter_adjuncts = False

# Representing complex heads
image_parameter_head_chains = True
image_parameter_complex_heads = False
image_parameter_covert_complex_heads = False
image_parameter_phrasal_complex_heads = False
image_parameter_draw_trivial_head_chains = False

# Phrasal chains
image_parameter_phrasal_chains = True

# DP-related
image_parameter_DP_hypothesis = False
image_parameter_DP_compression = False
image_parameter_shrink_all_DPs = False

# Overall properties of the chains
image_parameter_chain_curvature = 1.5

# Features
image_parameter_font = Times New Roman
image_parameter_features = R:NEW;R:OLD:REF;R:NEW:REF;ΦPF
image_parameter_visualization = R:NEW>r;R:OLD:REF>rflx;R:NEW:REF>pron;ΦPF>φ
```

Note that the output of the image generation can be modified and edited freely by using the GUI component. These settings are useful when the model is required to generate output images automatically for a whole dataset.

6.2.1.4 UG components

This section determines certain parameters of the underlying linguistic theory of grammar. Currently the user can run the simulation either with the standard or revised theory of Agree. In addition, the user can also determine the phase heads here.

```
# UG COMPONENTS
#   Agree
#   standard = standard theory (Chomsky 2000, 2001, 2008)
#   revised = current best formulation
UG_parameter_Agree = revised
Phi Level test = False

# Phase theory
UG_parameter_phase_heads = C;v;FORCE;Inf;IMPSS;P
UG_parameter_phase_heads_exclude = φ;v-;TO/inf

# Four conditions regulating word formation
head_complement_selection = True
epsilon = True
w_selection = True
UG_parameter_middle_field_HM = True

#   Semantics
calculate_assignments = True
calculate_pragmatics = False
calculate_thematic_roles = False
project_objects = True
generate_argument_links = True
calculate_predicates = True
calculate_operator_bindings = True
calculate_DIS_features = True
calculate_focus = True
```

6.2.1.5 Parsing heuristics

Parsing heuristics set a range of parameters that have to do with the heuristic principles the algorithm uses when attempting to find solutions for the input.

```
# PARSING HEURISTICS AND WEIGHTS
extra_ranking = True
filter = True
lexical_anticipation = True
closure = Bottom-up
working_memory = True
spec_selection = 100
comp_selection = 100
negative_semantics_match = 100
lf_legibility_condition = 100
```

6.2.2 Test corpus file (any name)

The test corpus file name and location are provided in the study configuration file. 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 as keys from the lexicon

exactly in the form they are provided in the input file. Depending on the research agenda a fully disambiguated lexicon could be considered an option.

Special symbols are used to render the 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. This allows the user to leave comments into the results file that would otherwise get populated with raw data only. An input sentences can be marked as ungrammatical by prefacing the it with an asterisk*. A sentence that has not been marked by * will be assumed to be grammatical. These markings should correspond to the gold standard/native judgments. 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). If the user wants to examine a group of sentences, they should all be prefaced with + symbol. 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 begin the processing in the middle of the file, use the symbol =START= (in effect, sentences between =START= and =STOP= will be processed).

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 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 (67).

(67) 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 (67)a did not end with the semicolon,

pronouns *he* and *her* (b) can no longer refer to them. Conversations can be used to create discourse contexts for test sentences.

6.2.3 Lexical files

The main script uses three lexical resource files that are by default called `lexicon.txt`, redundancy_rules.txt, `ug_morphemes.txt` and `inflections.txt`. The first contains language specific lexical items, the second a list of universal redundancy rules, universal morphemes and inflectional affixes. The number of lexical input files is not restricted, but they must be listed in the study configuration file.

6.2.4 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 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. 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. It is possible to use language specific redundancy rules. These are represented by pairing the antecedent feature with a language feature (e.g., [LANG:FI]).

6.3 Structure of the output files

6.3.1 Errors

A compilation of sentences with errors made by the model will be documented in the **error reports** file. If this file is empty, the model is observationally adequate.

6.3.2 Results

The name and location of the results file is determined when configuring the main script by the study configuration file. 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 list of study parameters used, time of the execution and locations of the input files. This followed by a grammatical analysis and other information concerning each example in the test corpus, each provided with a numeral identifier. What type of information is visible depends on the aims of the study.

6.3.3 The derivational log file

The derivational log file, created by default by adding `_log.txt` 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. By default 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.

6.3.4 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.

6.3.5 Resources file

The algorithm records the number of computational operations and CPU resources (in milliseconds) consumed during the processing of the first solution (§ 5.4). 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.

6.3.6 Phrase structure images

It is possible to examine the output of the algorithm in the form of phrase structure images. This functionality is currently available through the graphical interface, see Chapter 9.

6.4 Inputs, outputs and the graphical user interface

It is possible to examine and analyze the inputs and outputs by using a graphical interface. This functionality is explained in Chapter 9.

7 Grammar formalization

7.1 Basic grammatical notions (phrase_structure.py)

7.1.1 Introduction

The phrase structure class defined in the `phrase_structure.py` module implements phrase structure objects called *constituents* that are manipulated at each stage of the processing pipeline (PF-interface, LF-interface, partial representations produced by cyclic and noncyclic transformations).

7.1.2 Phrase structure geometry

The *daughter constituents* of any constituent α are provided as a list [X Y] such that α becomes the *mother* of both daughters. It is assumed that there can be only two daughter constituents, the *left* and the *right*, which are defined as follows:

```
def L(X):
    if X.const:
        return X.const[0]

def R(X):
    if X.const:
        return X.const[-1]
```

The left constituent is therefore always the first constituent in the list, the right the last. The left and right constituents are understood to define intrinsic ordering between them: the left constituent is ordered before the right constituents, such that the former precedes the latter, "precedence" referring to temporal order of neuronal processing that is often mirrored in the temporal processing of the surface sentence. The constituency relations do not define containment relations such that the higher order constituent would literally contain its daughters; the

asymmetry is an external property of the constituents that ties them together. If the left and right constituents are the same, that constituent is defined as an *affix*:

```
def affix(X):
    if X.L() == X.R():
        return X.L()
```

This condition becomes true if there is only one constituent. If α has three or more constituents (possible in theory), all other constituents except the first and the last will be "invisible."[59] The following functions determine whether X is left or right constituent of some host, where M(x) provides the mother of X (if any)

```
def is_L(X):
    return X.M() and X.M().L() == X

def is_R(X):
    return X.M() and X.M().R() == X
```

Most recursive operations which process constituents of the type [X Y] target [X Y] before targeting X and Y. This means that if [X Y] has independent content C that can be produced, C will be produced before X and Y. This situation occurs in the case of affixes: under [$_X$ Y] (usually notated as X(Y)), X will be spelled out before Y. This shows that the field `const` could be named `next` such that it points to the next element produced after its host, creating what amounts to a linked list of behaviors overly visible in morphology. Generalizing, [X Y] will be produced from left to right in the order [X Y] > X > Y where [X Y] is usually phonologically null but does not need to be, under current assumptions. In reality, then, the daughter constituents are elements produced in the specified order after their host, while the host is typically phonologically null. If these properties are not part of the human language faculty, then the implementation is too permissive and should be replaced with a narrower system.

---

[59] It is also possible in theory to posit daughter constituents which do not have mothers (i.e., the mother node is null). Neither option is used at present.

In some earlier versions the asymmetry was stipulated by positing separate left and right constituents (i.e. `self.left`, `self.right`), but this complicates many operations which have to process both constituents and which can now be defined as iterations over the list representation. In addition, there is no evidence that the left and right constituents differ in any other respects than in their order.

Terminal constituents do not have constituents, hence they are defined as not specifying anything that follows, while *complex* constituents have more than one (or alternatively `L(x) ≠ R(x)`):

```
def terminal(X):
    return not X.const

def complex(X):
    return len(X.const) > 1
```

The latter definition is suspicious since it assumes that grammar can count; an alternative is `L(x) ≠ R(x)` with both non-null. The notion of "complex constituent" seems artificial and should ultimately be deleted from the model if possible. A zero-level constituent has less that two constituents or it is defined as a zero-level by stipulation; the latter option is used only in the image generation algorithm and can be ignored here.

```
def zero_level(X):
    return len(X.const) < 2 or X.phrasal_zero_level
```

The notion of zero-level (=terminal or complex head) refers to something whose internal structure is outside of the domain of what is usually understood as phrasal syntax.

7.1.3 External and internal paths

7.1.3.1 General properties of search

Most of the grammatical relations and operations are defined by relying on *external* and *internal search* (§4.6). In a computational grammatical model every relation must ultimately rely on a search operation of some type that in one way or another traverses through phrase structure

geometry. If such search is an unavoidable, then the rest of the system should be reduced to a few fundamental and axiomatically defined search algorithms.

Currently the model has two fundamental search algorithms, one defining *internal paths*, another *external paths*, although the two have similar properties that suggests that a further reduction might be possible but not attempted here. Both search algorithms explore the contents of the syntactic working memory (i.e. linguistic content that has not been transferred out in phases), but they differ in the direction of search: internal search looks inside the constituent in a top-down manner, while external search looks outside into its grammatical context in a bottom-up direction. The latter, in particular, replaces the "feature vector" approach of my thesis (Salo, 2003). The former is a generalization of the notion of "minimal search" that played a central role in several previous versions of the model, and which was originally inspired by (Chomsky, 2008) and related work.

The search algorithms, whose implementation will be discussed below, work intuitively as follows. Internal search goes inside the target constituent by following labeling and selection, thus it follows XP…XP…XP until it encounters X^0, after which it enters into the sister phrase selected by X^0 if any, after which the process repeats until there are no more constituents. The external search follows dominance relations (forming the *spine* of the search) and detects daughter constituents (*branches*) that are not part of the spine. Both search algorithm thus defined are first-order Markovian operations. It is possible to provide internal search with a parameter which forces it to perform a recursive search called *scan*. Under this model, the search algorithm examines recursively all constituents inside XP. It is unclear if this operation should be defined as part of internal search, an alternative is to define its own function. Both search functions are wrapped inside entry functions currently called EXT and INT.

7.1.3.2 External search

External search begins by setting up default values and other data structures needed for the operation.

```
# Default values and preparations

x = X
y = None
collection = []
intervention = kwargs.get('intervention', lambda x: False)

# default value for domain is "top"

if kwargs.get('domain') == 'max':
    intervention = lambda x: x.M().head() != x.head()

# default value for criteria is "everything"

criteria = kwargs.get('criteria', lambda x: True)
if kwargs.get('self', False):
    collection.append(X)
```

Variables `x` and `y` are explained below. `collection` is a list that will contain elements encountered in the search path and will be returned to the caller. This can be thought of the nodes that will be under "attention" in the syntactic working memory when the search is executed; it provides the rest of the code access to those nodes. Both `intervention` and `criteria` are provided with the default values unless they are defined in the function call. `intervention` terminates search if the lambda-function applies to a node encountered during the search (default = no intervention, search as long as there are nodes) and defines the locality conditions for the search.[60] There is an optional argument `domain` which can currently be set to `max` and which provides an intervention function which restricts the search to the maximal projection. This makes the function calls more readable. `criteria` determines which nodes are included in the collection returned to the user (default = every node). If `self` is set True by the function call, then the target constituent will be included in the set of objects in the path, otherwise it is not. The initial steps are followed by the upward path loop:

[60] Because intervention is provided as a lambda-function, it is not limited in any way. It is currently unknown what general properties or principles determine the halting conditions (locality properties). When they are known, intervention should be replaced with a principled explanation.

```
# Upward path

while x.M() and not intervention(x):
    y = x
    x = x.M()
    for z in x.const:
        if z != y and criteria(z) and not (z.is_R() and z.adjunct):
            collection.append(z)
```

The operation follows dominance relations (`x = x.M()`) and collects daughters which (i) do not belong to the spine, (ii) have the property defined by criteria and (iii) are not right adjuncts. The search stops until there are no more mother nodes or the intervention condition applies to a node in the spine. `x` maintains the current position in the path, `y` stores a record of the previous node to define (i). The function returns either the first element satisfying the criteria (`acquire = minimal`), the last item in the spine (`acquire = maximal`) or it returns the whole collection:

```
# Return

if kwargs.get('acquire', 'minimal') == 'minimal':
    return next(iter(collection), None)
if kwargs.get('acquire') == 'maximal':
    return x
return collection
```

7.1.3.3 Internal search

The implementation of internal search is more complex since this version includes also recursive search (scanning), thus the function `internal_search_path` is defined recursively. It could be simplified by defining scanning inside its own function.[61]

The working memory collection that holds a record of the nodes on the search path is defined as a global class variable and gets filled by nodes if the `collect` parameter is set `True`:

---

[61] The main motivation for defining scanning inside internal search is because it is a form of internal search. The obvious counterargument is that as it is defined now the search function is divided into two separate operations, recursive scanning and first-order Markovian minimal search. The former in essence ignores the notion of syntactic working memory.

```
# Collect

if kwargs.get('collect', False):
    if kwargs.get('geometrical', False):
        PhraseStructure.speaker_model.syntactic_working_memory.append(X)
    else:
        if X.complex():
            PhraseStructure.speaker_model.syntactic_working_memory.append(X.L())
        else:
            PhraseStructure.speaker_model.syntactic_working_memory.append(X)
```

The global variable is used because the function is recursive. Geometrical search (`geometrical=True`) explores the right edge and stores (if set by `collect`) the right edge nodes. If either the spine or branch node matches with the criteria, the recursion begins to exit:

```
# Match

if X.match(NodePic):
    return X

if X.complex() and X.L().match(NodePic):
    return X.L()
```

Otherwise, if the node is complex, search continues. First we check if the spine or branch node satisfies intervention. If not, we implement scanning if it is set True (`scan=True`). Scanning means that we call the internal search path function recursively for both left and right constituents.

```
# Continue search (either X is complex OR we search inside comple

if X.complex() or (X.affix() and kwargs.get('affixes', False)):

    # Intervention

    if kwargs.get('intervention', lambda x: False)(X.L()) or \
            kwargs.get('intervention', lambda x: False)(X):
        return

    # Scan

    if kwargs.get('scan'):

        if kwargs.get('scan', False):
            Y = X.L().internal_search_path(NodePic, **kwargs)
            if Y:
                return Y
            return X.R().internal_search_path(NodePic, **kwargs)
```

If scanning is not activated, we use either *geometrical search* (search the right edge) or *minimal non-geometrical search*:

```
# Minimal search

else:

    # Geometrical search

    if kwargs.get('geometrical', False):
        return X.R().internal_search_path(NodePic, **kwargs)

    # Non-geometrical search

    if not X.R().adjunct:
        return X.R().internal_search_path(NodePic, **kwargs)
    else:
        return X.L().internal_search_path(NodePic, **kwargs)
```

The latter goes inside the right node unless it is an adjunct, in which case the search continues into the left constituent. The difference between geometrical and minimal search is that the latter dodges right adjuncts. Geometrical and minimal search could be implemented by first-order Markov loops like external search, but were not implemented as such due to scanning which relies on full recursion. An alternative is to factor geometrical and minimal search into one function defined by first-order Markovian loop and have a third function for scanning (or to find a way to eliminate scanning from the theory).

The path is stored into the global class variable (`PhraseStructure.speaker_model.syntactic_working_memory`) and can then be read from there by external functions, thus the function does not return collections which would make the handling more complex at the callers' side. However, the global class variable approach is not considered empirically implausible and reflects the idea that the search path makes certain nodes "visible" for other processes inside the human language faculty; the global class variable models this type of "visibility."

7.1.4 Sisters

Two constituents are *geometrical sisters* if they have the same mother. This notion can be defined by relying on external search (which itself relies on the mother-of dependency):

```
def geometrical_sister(X):
    return X.EXT()
```

The function, when called without arguments, returns the first constituent in the external search path, hence sister = closest object. Regular sister ignores adjuncts:

```
def sister(X):
    return X.EXT(criteria=lambda x: not x.adjunct)
```

This definition captures the assumption that adjuncts exists in a parallel working space. `EXT` is the wrapper function for external search (§7.1.4).

7.1.5 Specifier, complement and proper complement.

The notions of specifier, complement and proper complement are defined by relying on external search:

```
def complement(X):
    return X.EXT(domain='max')

def proper_complement(X):
    return X.EXT(domain='max', criteria=lambda x: x.is_R())

def specifier(X):
    return X.EXT(domain='max', criteria=lambda x: x.is_L())
```

A *complement* is the first constituent in the external path inside the projection from the head; *proper complement* has the additional property that it must be right; *specifier* is the first left constituent inside the projection. Because a primitive left constituent will always project (e.g., [$_{XP}$ X^0 Y(P)]), it can never be a complement or specifier. Under [X Y(P)] Y(P) will always be a complement and a proper complement. The complement and proper complement differ that the former extends to (i) YP in [$_{XP}$ YP [X ZP]] iff ZP is an adjunct and to (ii) XP in [$_{XP}$ YP X]. YP in (ii) is also a specifier, hence there are cases where a specifier is also the complement. These definitions exclude the possibility of multiple specifiers, but the formalism does not exclude [$_{XP}$ ZP [$_{XP}$ YP X(P)]]] where ZP is a non-adjunct; such ZPs are just not included under the definition of specifier above. We could relabel the above definition as “local specifier” and then defined a broader category of specifiers which include all left constituents inside the projection.

7.1.6 Heads (labels)

The head of any constituent is the first zero-level object encountered by internal search:

```
def head(X):
    return X.INT()
```

The core of the labeling algorithm elucidated in §4.5 is included in the definition of internal search. It returns the first or most prominent primitive constituent of XP.

7.1.7 Selection

Both complement and specifier selection are defined by external search. The current implementation defines both positive and negative specifier and complement selection, although negative selection is superfluous and only exists to make the selection more easy to understand in those cases when it is relies principally on negative selection ('select anything but…'). Both selection features have the form [±COMP/SPEC:F1…FN] where ± represents polarity, COMP/SPEC defines the configuration and $F_1 \ldots F_N$ is an exclusive feature list such that everything else is assumed to belong to the opposite category. For example, all features not listed in a positive complement feature are assumed to be specified for negative selection. If the lexicon contains two separate selection features, they are synthesized into one feature during lexical retrieval. The ability to have a null complement or specifier must be represented by a special feature which is ø in the current implementation. [+SPEC:N, Ø] means that the lexical item selects N-specifiers but is also licensed to remain without a specifier.

First we define abstractions for the notions of specifier, complement and proper complement:

```
def complement(X):
    return X.EXT(domain='max')

def proper_complement(X):
    return X.EXT(domain='max', criteria=lambda x: x.is_R())

def specifier(X):
    return X.EXT(domain='max', criteria=lambda x: x.is_L())
```

All definitions rely on external search. Having these functions at hand we can define negative and positive specifier and complement selection as follows:

```
def minus_SPEC(X, fset):
    Y = X.specifier()
    return not Y or not Y.head().INT(some(fset))

def plus_SPEC(X, fset):
    Y = X.specifier()
    return (not Y and 'ø' in fset) or (Y and Y.head().INT(some(fset)))

def plus_COMP(X, fset):
    Y = X.proper_complement()
    return (not Y and 'ø' in fset) or (Y and Y.head().INT(some(fset)))

def minus_COMP(X, fset):
    Y = X.proper_complement()
    return not Y or not Y.head().INT(some(fset))
```

First we target the appropriate syntactic object Y, specifier or complement, depending on the type of selection. Then we examine if the head of Y contains the selection features `fset`. As pointed out above, negative selection can be eliminated from the theory, but remains in the current model to make lexical behavior more explicit. It is also possible to test selection against the head itself. This is called *self-selection*:

```
def minus_SELF(X, fset):
    return not X.INT(some(fset))

def plus_SELF(X, fset):
    return X.INT(some(fset))

def conjunctive_minus_SELF(X, fset):
    return not X.INT(fset)
```

Self-selection is used to test that the head occurs in an appropriate state at the LF-interface. For example, we can create filters which make sure that all heads undergo certain operations by verifying their feature content.

The model contains a nonlocal selection option based on *probing*. Feature [±PROBE:F] finds the first zero-level constituent with feature F by internal search (minimal search) from the sister of the triggering head:

```
def probe(X, G):
    return X.sister().INT(criteria=lambda x: x.zero_level() and x.INT({G}),
                          intervention=lambda x: x.INT('finite_C'))
```

The notation `x.INT({G})` means that we search for the features in the set {G}, here only one feature, but in principle it is possible to search for several features conjunctively.

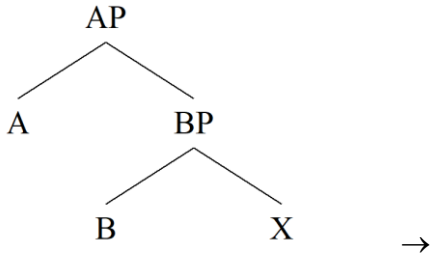
7.1.8 Structure building

Structure building refers to the computational assembly operations which create new structure by merging phrase structure constituents together. In this version the core assembly operations are defined by relying on Python "magic methods" which have the advantage that the source code can be made to mimic the empirical theory. For example, the bare external Merge operation which combines two existing phrase structures X and Y into [X Y] is defined originally by relying on the `__init__` function of the phrase structure class which can take the two constituents are input arguments and then by embedding this construct inside Python's `__add__` magic method

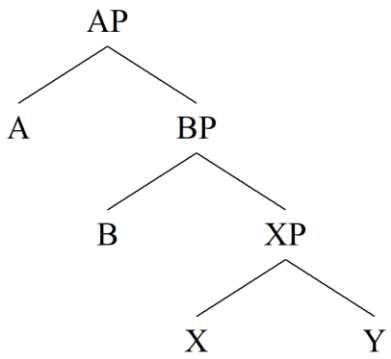
```
def __add__(X, Y):
    # Standard bare Merge
    return PhraseStructure(X, Y)
```

We can now refer to the operation in the code by +: `Z = X + Y` at the level of code means `Y = PhraseStructure(X, Y)`.

Merge-1 is a slightly more complex operation: it targets an existing node X, merges the other constituent Y either to its right or left, and the substitutes the original X with the new constituent Y + X in the phrase structure:



→



The definition for Merge-1 is part of `__mul__` which refers to multiplication at the level of code.

```
def __mul__(X, Y):

    # Insert Merge (i.e. Merge-1)

    X.consume_resources('Merge-1', X)
    if X.M():
        return X.insert(Y, 'right')
    elif Y.M():
        return Y.insert(X, 'left')
    return X + Y
```

If both X and Y are external constituents without existing mothers, hence they do not have a mother, the operation performs bare external X + Y (last line). That is, Merge-1(X, Y) = [X Y] if both X and Y are primitive. If X has a mother (as in the phrase structures above), then Y is merged-1 to its right (as above): X * Y with […X…] creates […[X Y]…]. The two operations are used during the left-to-right cycle. If Y has mother, then X is added to its left: X * Y with […Y…] creates […[X Y]…]. This option is used during transfer, for example when reconstructing heads. Function `insert` implements the actual insertion, which in effect substitutes the original X inside […X…] with [X Y]:

```
def insert(Xa, Y, dir=''):
    if Xa.is_L():
        return Xa.M().insert_left(Xa.Merge(Y, dir))
    elif Xa.is_R():
        return Xa.M().insert_right(Xa.Merge(Y, dir))
    return Xa.Merge(Y, dir)


def insert_left(X, Y):
    return X.create_constituents([Y, X.R()]).L()

def insert_right(X, Y):
    return X.create_constituents([X.L(), Y]).R()
```

`Merge` creates the new constituent [X Y], which is then substituted for the original X by `insert` depending on whether the original was left or right constituent: […X…] → […[X Y]…].

```
def Merge(X, Y, dir):
    if dir == 'left':
        return Y + X
    return X + Y
```

Substitution involves updating both the daughter and mother dependencies. It is likely that these operations can be simplified, though this possibility remains to be studied.

The exponent function ** is defined as an operation which adds Y inside X as an affix:

```
def __pow__(X, Y):
    # Head merge (HM)
    return X.affixes()[-1].create_constituents([Y])
```

Thus, X ** Y = X(…Y) where Y will be the bottom affix inside X.

7.1.9 Tail-head relations

A tail-head dependency is formed when a probe must locate a target G either inside the head of its own projection ("strong tail test") or inside an upward path by memory scanning (21)("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(X, **kwargs):
    tail_sets = kwargs.get('tail_sets', X.get_tail_sets())
    if not tail_sets:
        return True
    weak_test = kwargs.get('weak_test', True)
    direction = kwargs.get('direction', 'left')
    positive_tsets = {frozenset(positive_features(tset)) for tset in tail_sets if positive_features(tset)}
    negative_tsets = {frozenset(negative_features(tset)) for tset in tail_sets if negative_features(tset)}
    checked_pos_tsets = {tset for tset in positive_tsets if X.tail_condition(tset, weak_test, direction)}
    checked_neg_tsets = {tset for tset in negative_tsets if X.tail_condition(tset, weak_test, direction)}
    return positive_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(X, tail_set, weak_test, direction):
    if weak_test:
        for x in X.EXT(acquire='all', criteria=lambda x: x.zero_level() and
                                                         x.is_L(), self=(direction == 'right')):
            if x.INT(some(tail_set)):
                return x.INT(tail_set)
    else:
        return X.max().container() and X.max().container().INT(tail_set)

def get_tail_sets(X):
    return {frozenset(f[5:].split(',')) for f in X.head().core.features() if f.startswith('TAIL:')}
```

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 second clause and essentially tests that the tail features are present in the head of the projection inside of which the testing element is. 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.

7.2 Transfer

7.2.1 A comment on the current version (20.1)

The organization and operation of the transfer function has undergone major changes and simplifications. As discussed in §4.8, transfer is composed of several operations such as head reconstruction, A-chains, Ā-chains and scrambling. Many earlier models treated each reconstruction operation as its own class and utilized functions that were specific to that operation. It was impossible to calculate any realistic data without using specialized modules for each operation. On the other hand, it was clear from the beginning that the operations were also based on a general template. 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 defined elsewhere remain.

Transfer was also dissolved into cyclic and noncyclic operations. Cyclic operations apply as soon as an error configuration is detected during the initial left-to-right assembly. Noncyclic operations apply to completed phrase structures, both left phrases and final candidate representations, during transfer. Some operations apply both cyclically and noncyclically.

Properties of transfer are partitioned into two blocks, one representing the general or common features of all transfer operations, another which represents what is idiosyncratic for each particular operation. The latter are represented in one dictionary table `operations` currently coded as a class variable for phrase structure class, the former is presented by two general functions which refer to the table in their operation. The main goal is to reduce the idiosyncratic component into its bare essentials while expanding the general part. This process has to be performed in the context of a representative dataset.

7.2.2 General transfer functions (transfer and reconstruct)

Transfer is defined by two functions, `transfer`, which applies noncyclic reconstruction, and `reconstruct`, which takes some node as input and applies a set of reconstruction operations to it. The former calls the latter, but the latter is also called when performing cyclic reconstruction during the left-to-right derivation.

Let us consider `reconstruct` first.

```
343     def reconstruct(X, **kwargs):
344         """
345         Reconstruction cycle for element X (head or phrase).
346         """
347
348         # Get the operations from an input parameter or assume they apply all
349
350         OPs = kwargs.get('operations', PhraseStructure.operations)
351
352         # Apply each operation to X
353
354         for type, calculate in OPs.items():
355             if not X.copied and calculate['TRIGGER'](X):
356                 T = calculate['TARGET'](X)
357                 if T and not T.copied:
358                     log(f'\n\t{type}({T.illustrate()})')
359                     X = calculate['TRANSFORM'](X, T).new_focus()
360                     log(f'\n\t = {X.top()}')
361
362         return X.top()
```

It is possible to call this function by providing a dictionary of operations that are applied to the target node X (line 350). If no dictionary is provided, all operations, which are defined in a separate table (§7.2.3), are applied. The function applies each operation (line 354) to the node X. Each operation consists of three separate steps: `TRIGGER`, which examines if the operation is triggered for X, `TARGET`, which determines the target of the operation (usually X or its Spec) and `TRANSFORM` which implements the reconstruction. Trigger will detect if X or XP has some "defect" that needs repair; target will determine the constituent that must be changed, while Transform fixes the issue. A crucial feature of this function is that all operations are applied in a well-defined sequence (as defined by the dictionary) to X.[62] The function `new_focus` determines a possible new X after some operation has been implemented, since the structure may have changed due to a previous operation.

Function `transfer` is provided below:

[62] We are assuming a version of Python which provides dictionaries (or rather their keys) with a well-defined ordering.

```
def transfer(X):
    Y, m = X.detach()

    # Reconstruction applies noncyclically during transfer such
    # each operations applies to the whole structure,
    # and it applies successive-cyclically until no further stru

    for type, calculation in PhraseStructure.operations.items():
        size = 0
        while Y.top().size() != size:           # Apply each ope
            size = Y.top().size()               # Store size bef
            for x in Y.bottom().EXT(acquire='all', self=True):
                x.reconstruct(operations={type: calculation})
    return Y.reattach(m)
```

This function goes through all reconstruction operations and applies them to all nodes in X in a bottom-up cycle. Each operation therefore constitutes its own cycle, and evaluates the whole expression from bottom-up up. The nodes are determined by external search path (§7.1.3.2). The whole cycle is repeated until there is no change. This implements successive-cyclicity. Transfer, as the new implies, implements noncyclic transfer: it applies to the whole structure when it is transferred to LF; however, it uses reconstruct described above.

7.2.3 Reconstruction operations (PhraseStructure.operations)

The operations table provides instructions for each individual reconstruction operation by specified four fields `TRIGGER`, `TARGET` and `TRANSFORM` that were explained in the previous section.

```
operations = {'Noncyclic Ā-chain':
                {'TRIGGER': lambda x: x.operator_in_scope_position() and
                                      not PhraseStructure.cyclic,
                 'TARGET': lambda x: x,
                 'TRANSFORM': lambda x, t: x.reconstruct_operator(t)},
              'Feature inheritance':
                {'TRIGGER': lambda x: x.zero_level() and x.INT(['φ', 'EF?', 'Fin']),
                 'TARGET': lambda x: x,
                 'TRANSFORM': lambda x, t: x.feature_inheritance()},
              'A-chain':
                {'TRIGGER': lambda x: x('EPP') and
                                      x.is_R() and x.sister() and x.sister().complex() and
                                      x.sister().INT('referential') and not x.sister().INT('operator', scan=True) and
                                      x.tail_test(tail_sets=x.sister().get_tail_sets(), direction='right', weak_test=True),
                 'TARGET': lambda x: x.sister().chaincopy(),
                 'TRANSFORM': lambda x, t: x * t},
              'IHM':
                {'TRIGGER': lambda x: x.complex_head() and not x('EHM'),
                 'TARGET': lambda x: x.affix(),
                 'TRANSFORM': lambda x, t: x.head_reconstruction(t)},
              'Scrambling':
                  {'TRIGGER': lambda x: x.max().license_scrambling() and
                                        (x.max().container() and x.max().container()('EF') and (not x.max().container()('thet
                                        and x.max().scrambling_target() and x.max().scrambling_target() != x.top() and not x.
                                        PhraseStructure.speaker_model.LF.pass_LF_legibility(x.max().scrambling_target().copy(
                                        not PhraseStructure.cyclic,
                   'TARGET': lambda x: x.max().scrambling_target(),
                   'TRANSFORM': lambda x, t: x.max().scrambling_reconstruct(t)},
              'Agree':
                  {'TRIGGER': lambda x: x.sister() and x.is_L() and x.core.features(type=['phi', 'unvalued']) and not x('fini
                   'TARGET': lambda x: x,
                   'TRANSFORM': lambda x, t: x.AgreeLF()}
                   }
```

The table shows that while most of the operations are already defined by relatively simple functions, this is not true of scrambling ("free word order"). These conditions also define whether the operation is applied cyclically, noncyclically or both. Currently Ā-reconstruction and scrambling reconstruction are noncyclic, the rest apply during both. Ideally they will all apply freely during both, but this still provides wrong predictions in some cases and was not assumed here. The operations are applied in the order determined by the dictionary (from top to down), and this order has empirical consequences and constitutes an important component of the model.

7.2.3.1 Ā-reconstruction

Ā-reconstruction applies when an operator X(P) exists in a left peripheral scope position (`operator_in_scope_position`) at the noncyclic portion of the derivation (Trigger). It applies to X(P) (Target) and reconstructs (copies) it into a thematic base position (Transform)(68).

(68) *Who* did John meet __?

'which x: John met x'

The triggering condition is defined as follows:

```
def operator_in_scope_position(X):
    """
    An operator in scope position is one which requires Ā-reconstructions
    (1) it must be an operator
    (2) it must not be a scope-marker
    (3) either it is contained in SpecCP or it is a operator predicate
    """
    return X.INT('operator', scan=True) and \
           not X.core.features(match=['$OP$']) and \
           ((X.container() and X.container().INT({'Fin'})) or
            (X.INT({'-insitu'}) and X.INT(['TAM', 'C/fin', 'Neg/fin'])))
```

The last condition (3) implements reconstruction for Finnish long head movement (predicate clefting), thus they are treated like regular Ā-operators and are not reconstructed by IHM (§7.2.3.4)(69).

(69) Myy-dä-kö Pekka aikoo __ kaiken omaisuutensa?

sell-A/INF-Q Pekka plans all possessions

'Was it selling x: Pekka plans to do x for all his possessions'

The operation performs internal search (§7.1.3.3) into the sister of the operator and tries to find a thematic base position (line 530). It dissolves into two separate blocks, one which handles the local Aux-inversion/V2 data (*did John __ admire Mary?*) and another which reconstructs in a nonlocal manner (*who did John admire __?*). The implication is that the former is considered a "special" operator construction. The former detects fronted auxiliaries and reconstructs them locally, the latter reconstructs into closest vacant thematic positions.

```
def reconstruct_operator(X, T):
    for x in X.sister().collect_sWM(intervention=lambda x: x.zero_level() and x.INT('referential')):

        #   Sentence operator, null head, V2 (local X-to-C)

        if T.zero_level() and \
                T.INT('finite') and \
                not T.EXT(criteria=lambda x: x.zero_level() and x.INT('finite_C')):

            T = X.chaincopy()
            if x.complex():
                return T * x.sister()       # [T [XP [T' YP]]], XP = x
            return T * x                    # [T [T' [K YP]]], K = x

        # Found a position where tail tests succeed

        if x.tail_test(tails_sets=T.get_tail_sets()) and x.zero_level():

            if T.complex():

                # New SPEC

                if not x.specifier() and T.INT(some(x.core.get_selection_features('+SPEC'))):
                    if x.is_R():
                        return T.chaincopy() * x
                    return T.chaincopy() * x.M()

                # New COMP

                if T.INT(some(x.core.get_selection_features('+COMP'))):     # (2.1) No complement =
                    if not x.complement():
                        return x * T.chaincopy()

            elif T.zero_level():

                # LHM

                if T.INT(some(x.core.get_selection_features('+COMP'))):
                    return T.chaincopy() * x.sister()
    return X
```

We find positions that can check tail-features/case and selection. LHM is implemented by the last block.

7.2.3.2 Feature inheritance

Feature inheritance changes the feature content of heads based on their grammatical context and applies to three cases currently: imposing special properties to the highest finite node to account for special EPP/Agree properties; regulating the absence and presence of overt argument

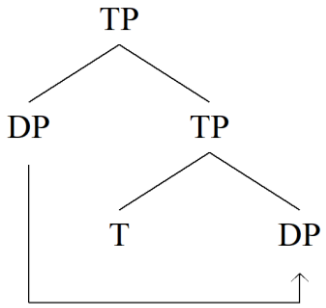
depending on selection, accounting for obligatory control and other similar properties; creating phi-feature concord inside nominal projections.

```
def feature_inheritance(X):

    # C-T feature inheritance

    if X('finite') and \
            not X.EXT(criteria=lambda x: x.zero_level() and x('finite')) and not X.INT({'!PER'})
        X.core.add_features({'!PER'})

    # Obligatory control

    if X.INT({'EF?'}):
        X.core.remove_features({'EF?'})
        X.core.add_features({'EF*'})
        if X.EXT(criteria=lambda x: x.zero_level() and x.INT('SEM_internal_predicate')):
            X.core.add_features({'-ΦLF'})

    # DP-internal concord (rudimentary, not studied in detail)

    if X.INT({'φ'}) and X.complement() and X.complement().is_R():
        X.core.add_features(X.complement().head().core.features(type=['phi', 'interpretable']))
        X.core.add_features(X.complement().head().core.get_R_features())
    return X
```

The operation is triggered by the presence of relevant features and changes the feature content accordingly. The concord algorithm is basic and not explored in any detailed study; it is likely that all feature inheritance operations will reduce to one more general concord mechanism, but testing anything more specific requires a representative and specifically curated dataset that contains expressions exemplifying all the relevant phenomena.

7.2.3.3 A-reconstruction

A-reconstruction applies to standard EPP-heads under [YP X] where YP is a referential non-operator argument, X is an EPP head and when YP could check its case against X. Most of the application happens cyclically due to the triggering configuration [YP X] that rarely exists (but could exist) during the noncyclic stage. YP is copied and merged to [$_{XP}$ YP [$_{XP}$ X ~~YP~~]]. The operation itself is fully specified in the operations table which creates [$_{XP}$ X YP] by X * YP. A prototypical case is the early portion of English finite clause:



7.2.3.4 IHM (Internal head merge)

IHM (internal head merge) corresponds to standard head movement/reconstruction and applies if X is a complex head and is not created by external head merge (EHM). Cyclic application is blocked if the complex head is an operator; in that case it will be reconstructed by noncyclic Ā-reconstruction.

```
def head_reconstruction(X, T):
    if PhraseStructure.cyclic:
        if not X.is_L():    # X is right or isolated = cyclic IHM
            if not (X.INT('operator', criteria=lambda x: True, intervention=lambda x: False, scan=True) and
                    X.INT('-insitu', criteria=lambda x: True, intervention=lambda x: False, scan=True) and
                    X.INT('TAM', criteria=lambda x: True, intervention=lambda x: False, scan=True)):   #   E
                return X * T.chaincopy()
            return X
        else:   # X is left, special case of A-chain + IHM')
            return X.sister() * T.chaincopy()
    else:
        return T.chaincopy() * X.sister()
```

The cyclic transforms are (i) [$_{XP}$ ZP X(Y)] → [$_{XP}$ ZP [$_{XP}$ X Y]] and (ii) [$_{XP}$ X(Y) ZP] → [$_{XP}$ X [ZP Y]], where the latter is used after phrasal A-chain so that the full sequence is [$_{XP}$ YP X(Y)] → [$_{XP}$ YP [X(Y) YP]] (A-chain, §7.2.3.6) → [$_{XP}$ YP [$_{XP}$ X [YP Y]]] (head chain), the whole sequence is performed during one reconstruct call (hence, A-chain must precede IHM in the operators table). For example X = T, Y = v. The noncyclic transform is (iii) [$_{XP}$ X(Y) ZP] → [$_{XP}$ X [$_{YP}$ Y ZP]] which implements local HMC-compliant head reconstruction. Prototypical examples of each are expanding V out of v (i); expanding V out of T in connection with A-chain (ii); expansion of any other residuum complex head not performed cyclically (iii).

7.2.3.5 Scrambling reconstruction

Scrambling reconstruction of XP applies noncyclically if the tail-features of XP cannot be checked and it is not an operator. It reconstructs XP into a thematic position where the tail-features

can be checked. The function will also create adjuncts when necessary, in an operation called in-situ scrambling:

```
# In situ scrambling: if the externalized XP is in correct position, leave it there

if XP.INT('adverbial') or XP.INT('preposition'):
    weak_test = False
else:
    weak_test = True
if XP.tail_test(weak_test=weak_test) and not (XP.container() and XP.container().INT('EF')):
    return XP.head().top()
```

The target phrase was earlier turned into adjunct and here tested if it could be in an acceptable position and, if it is, the nothing else will be done. Otherwise, we search for a new position:

```
# Search for a new position

for x in XP.local_tense_edge().collect_sWM(intervention=lambda x: 'φ' in x.core or x == XP.head()):

    # Specifier positions

    if x.zero_level() and x.tail_fit(YP) and YP.spec_selection(x):
        if O.max().container() == x and Spec:
            continue
        if x.is_L():
            return YP * x.M()     #   [a X(P)] = [YP [a X(P)]]
        return YP * x             #   [X(P) a] = [X(P) [YP a]]

    # Complement positions

    if x.zero_level() and not x.proper_complement() and x.tail_fit(YP, 'right') and x.comp_selection(YP):
        return x * YP             #   [X(P) a] = [X(P) [a YP]] or [a <XP>] = [[a YP] <XP>]
```

Internal search from the minimal tense is used, we search for suitable specifier and complement positions. If nothing is found, we do not reconstruct anything.[63]

7.2.3.6 Agree

Agreement reconstruction applies if X has unvalued phi-features (i.e. it is a predicate). It applies standard Agree to X. The function finds a suitable goal and values its phi-features to X:

```
def AgreeLF(X):
    return X.Agree(X.get_goal())
```

Function `get_goal` finds a possible goal from within the sister of X.

---

[63] A possible simplification is to write the specifier and complement conditions as a criteria for the internal search, but this is not yet done in order to keep the function as clear as possible.

```
def get_goal(X):
    return X.sister().INT(criteria=lambda x: not x.copied and x.head()('referential'),
                          intervention=lambda x: x.phase_head())
```

We search XPs which are referential (generally, DPs) but do not look past phase heads which are defined by its own function. The goal is sent to Agree

```
684     def Agree(X, goal):
685
686         # Agreement is executed if the probe has unvalued phi-features and there is a goal
687
688         if len(X.core.features(type=['phi', 'unvalued'])) > 0 and goal:
689
690             # Valuation is possible if there are no phi-feature mismatches between the probe and goal
691             # This handles standard agreement errors
692
693             if not goal.head().core.feature_mismatch_test(X.core.phi_bundles()):
694                 X.core.value(goal)
695             else:
696                 log(f' FEATURE MISMATCH {goal.head().core.feature_mismatch_test(X.core.phi_bundles())}')
697                 X.core.add_features({'*'})
698         else:
699             log(f' did not find suitable goal.')
700         return X
```

which performs valuation (line 694) unless it is blocked by conflicting features at the probe (line 693). The latter condition rules out sentences such as **We admires Mary*. Actual feature valuation is performed on line 216-217 inside function `value`.

```
206     def value(self, Y_goal):
207         log(' values ')
208
209         # get features from the goal
210
211         fset = self.features_to_value_from_goal(Y_goal)
212         if fset:
213
214             # Value the acquired features
215
216             for phi in fset:
217                 self.value_phi_feature(phi)
218                 log(f'[{phi[5:]}]')
219         else:
220             log(f'nothing')
221         log(f' from goal {Y_goal.max()}.')
222
223         # Partial agreement does not create ΦLF
224
225         if len(fset) > 1:
226             self.add_features({'ΦLF'})
227             self.remove_features({'?ΦLF'})
228         self.add_features({f'PHI:IDX:{Y_goal.head().core.get_id()}'})
```

This converts unvalued features such as [PHI:NUM:_] into valued features like [PHI:NUM:SG]. After valuation, feature ΦLF is added to the head to signal that it was valued by Agree. Line 228 adds a pointer to the goal. Feature mismatch tests are nontrivial, and currently no published studies exist. The problem is that the phi-features at the probe can be more abstract than those occurring at the goal. For example, in English -s signals third person singular whereas its absence signals

everything else; everything else is defined as a list of phi-features. First we (i) remove features between the probe and goal which match (line 280, the test in the *for*-loop), and then (ii) we verify that the residuum does not conflict (function `mismatch`).

```
269     def feature_mismatch_test(X, PP):
270         """
271         X = goal
272         PP = phi-bundles at the probe
273         This function examines if there are unlicensed phi-features at the goal (G) that mismatch with
274         phi-features at the probe. Unlicensed phi-features at the goal are those features which are not
275         matched with phi-bundles at the probe.
276         Note 1: The feature format is TYPE:VALUE with (i)PHI removed.
277         """
278
279         def unlicensed_phi_features_at_goal(G, PP):
280             return G - set().union(*(frozenset(phi) for phi in PP if phi <= G))
281
282         return mismatch(unlicensed_phi_features_at_goal(X.features(type=['phi', 'interpretable'], format='reduced'), PP), set().union(*PP))
```

For example, if the probe has features [NUM:SG] and [NUM:PL] (i.e. it abstracts from number) and the goal has [NUM:PL], then the function `unlicensed_phi_features_at_goal` returns all other features of the goal except [NUM:PL] that has now been "checked" for correctness (i.e. comparing plural goal against probe that is compatible with both singular and plural). These unchecked features are then checked against features at the probe for possible mismatches (e.g., [PER:1] vs [PER:2]).

7.3 LF-legibility (LF.py)

7.3.1 Overall

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

```
all_legibility_tests = [('Selection test', self.selection_test),
        ('Head Integrity test', PhraseStructure.unrecognized_label),
        ('Feature Conflict test', PhraseStructure.feature_conflict),
        ('Probe-Goal test', PhraseStructure.probe_goal_test),
        ('Semantic Complement test', PhraseStructure.semantic_selection),
        ('EPP test', PhraseStructure.EPP_violation),
        ('Core integrity', PhraseStructure.core_integrity),
        ('External head merge test', PhraseStructure.complex_head_integrity),
        ('Projection Principle', PhraseStructure.projection_principle_failure)]
return [test for test in all_legibility_tests if self.speaker_model.settings.retrieve(test[0], True)]
```

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. 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.

```
def pass_LF_legibility(self, X, **kwargs):
    self.logging = kwargs.get('logging', True)
    self.failed_feature = ''
    if not X.copied:
        if X.zero_level():
            for (test_name, test_failure) in self.active_test_battery:
                if test_failure(X):
                    if self.speaker_model.settings.retrieve('dev_logging', False):
                        self.speaker_model.settings.application.dev_logging(f'\n\tFailed {test_name}, {X.top().illustrate()})')
                    if self.logging:
                        log(f'\n\t{X} ({X.max().illustrate()}) FAILED {test_name} ')
                        if self.failed_feature:
                            log(f'for [{self.failed_feature}]')
                    self.error_report_for_external_callers = f'{X} failed {test_name}.'
                    return False
        else:
            if not self.pass_LF_legibility(X.L(), logging=self.logging):
                return False
            if not self.pass_LF_legibility(X.R(), logging=self.logging):
                return False
    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.

```
self.selection_violation_test = {'-SPEC': PhraseStructure.minus_SPEC,
                                 '+SPEC': PhraseStructure.plus_SPEC,
                                 '-COMP': PhraseStructure.minus_COMP,
                                 '+COMP': PhraseStructure.plus_COMP,
                                 '-SELF': PhraseStructure.minus_SELF,
                                 '+SELF': PhraseStructure.plus_SELF,
                                 '=SELF': PhraseStructure.conjunctive_minus_SELF}
```

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

```
def selection_test(self, X):
    for key in X.core.selection_keys():
        if key in self.selection_violation_test.keys():
            if X.core.get_selection_features(key):
                if not self.selection_violation_test[key](X, X.core.get_selection_features(key)):
                    if self.speaker_model.settings.retrieve('dev_logging', False):
                        self.speaker_model.settings.application.dev_logging(f'\n\t{X} failed {key} {X.core.get_selection_features(key)}. ')
                    if self.logging:
                        log(f'\t{X} failed {key}: {X.core.get_selection_features(key)} ')
                    return True     # Failed test
```

The function activates a selection test if the lexical item has the gate feature, as listed in the selection test list above.

7.3.2 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.

```
def projection_principle_failure(X):
    """Tests if there are referential arguments that do not receive theta roles"""

    if X.max() != X.top() and X.projection_principle_applies():
        return not X.max().gets_theta_role_from(X.max().container()) or X.pro_projection_principle_violation()
```

The theta-role function is defined as follows:

```
def gets_theta_role_from(Xmax, Y):

    # Y assigns theta-role to SPEC

    if Xmax == Y.specifier() and Y.INT('thetaSPEC'):
        return True

    # Y assigns theta-role to COMP

    if Xmax == Y.proper_complement() and Y.INT('thetaCOMP'):
        return True

    # If neither above is true, Y can assign theta-role to left sister iff it is not EF-head

    if Xmax == Y.sister() and Y.INT('thetaCOMP') and not Y.INT('EF'):
        return True
```

It checks three conditions: whether `Xmax` is at Spec of `Y` and receives a thematic role from the head at that position; whether `Xmax` is the Comp of `Y` and receives a thematic role from the head at that position; and whether `Xmax` is a left sister of `Y` and receives a thematic role. The function `pro_projection_principle_violation`checks that pro-elements do not violate the projection principle.

```
def pro_projection_principle_violation(X):
    if X.zero_level() and X.core.overt_phi_sustains_reference() and X.complement():
        return X.complement().INT(criteria=lambda y: y.zero level() and y.core.overt phi sustains reference(),
```

The function performs minimal search into the complement of X and checks that it does not have another pro-element linked with a verbal predicate. The empirical function of this principle is to rule out expressions with full phi-agreement on several (usually auxiliary, negation) heads above their main verb.

7.4 Lexicon and the lexical core

Primitive phrase structure objects (heads) can contain lexical features, the latter which are inside the *core* of the head. The core is modelled by a separate class `PhraseStructureCore` in the module `phrase_structure_inner_core.py`. It is currently represented as a list of feature sets that can be created from sets or lists:

```
def __init__(self, **kwargs):

    # The core contains a list of feature sets (feature bundles)

    if isinstance(kwargs.get('features'), set):
        self._features = [kwargs.get('features', set())]
    elif isinstance(kwargs.get('features'), list):
        self._features = kwargs.get('features', [])
    else:
        self._features = [set()]
```

The list of feature sets are called *feature bundles*. Function `features` will return all features from the core in one set or, if provided with keywords such as *match* or *type* for searching, features which match criteria:

```
def features(self, **kwargs):
    fset = set().union(*[fset for fset in self._features])

    # We can probe features which match with strings and substrings
    # todo This will be implemented by regex pattern matching

    if kwargs.get('match'):
        fset2 = set()
        for f in fset:
            for pattern in kwargs.get('match'):
                if pattern.endswith('...'):
                    if f.startswith(pattern[0:-3]):
                        fset2.add(f)
                elif pattern.startswith('$') and pattern.endswith('$'):
                    if pattern == f:
                        fset2.add(f)
                elif pattern in f:
                    fset2.add(f)
        fset = fset2

    # We can also probe with features by referring to their type

    if kwargs.get('type'):
        for feature_type in kwargs.get('type', []):
            fset = {f for f in fset if feature_abstraction[feature_type](f)}
        if kwargs.get('format') == 'reduced':
            fset = {':'.join(f.split(':')[1:]) for f in fset}
        if kwargs.get('format') == 'no_value':
            fset = {':'.join(f.split(':')[0:-1]) for f in fset}
        if kwargs.get('format') == 'only_value':
            fset = {f.split(':')[-1] for f in fset}
        if kwargs.get('format') == 'licensed':
            fset = set().union(*{frozenset(f.split(':')[1].split(',')) for f in fset})

    return fset
```

This class provides simple interfaces also for adding, removing and setting the features. The intuitive function of the feature bundle system is to represent sequences of inflectional information and especially clitics. Thus, one head can contain several phi-bundles, each having a different function (representing arguments, agreement, concord) in the system. Each bundle is generated from the inflectional feature clusters in the order they appear in the input word.

Core-objects define function `__call__` with a string argument that allows one to execute queries and thus define classes of primitive heads:

```
def __call__(self, stri):

    # Safeguard

    if not stri:
        return True

    # If the string stri provides a valid key for the dictionary containing the definition,
    # test whether the definition applies

    if abstraction_funct.get(stri):
        return abstraction_funct[stri](self)
```

The `abstraction_funct` is a dictionary of keyword-lambda-function pairs which define the properties.

```
abstraction_funct = {'EHM': lambda X: 'ε' in X,
                     'EF': lambda X: ['EF', 'EF*'] in X,
                     'EPP': lambda X: 'EF*' in X,
                     'TAM': lambda X: X.features(match=['TAM']),
                     'event': lambda X: 'π' in X,
                     'polarity': lambda X: X.features(match=['POL:']),
                     'strong_features': lambda X: X.features(match=['**...']),
                     'predicate': lambda X: 'Φ' in X,
                     'finite': lambda X: ['Fin', 'T/fin', 'C/fin'] in X,
                     'force': lambda X: 'FORCE' in X,
                     'finite_C': lambda X: 'C' in X,
                     'referential': lambda X: ['φ', 'D'] in X,
                     'nonreferential': lambda X: 'SEM:nonreferential' in X,
                     'adverbial': lambda X: 'Adv' in X,
                     'adjoinable': lambda X: 'adjoinable' in X,
                     'nonfloat': lambda X: 'nonfloat' in X,
                     'nominal': lambda X: 'N' in X,
                     'verbal': lambda X: 'ASP' in X,
                     'theta_predicate': lambda X: X('thetaSPEC') or X('thetaCOMP'),
                     'thetaSPEC': lambda X: 'θSPEC' in X,
                     'thetaCOMP': lambda X: 'θCOMP' in X,
                     'nonthematic_verb': lambda X: X('verbal') and not X('theta_predicate'),
                     'thematic_edge': lambda X: X.get_selection_features('+SPEC') & {'D', 'φ'},
                     'light_verb': lambda X: ['v', 'v*', 'impass', 'cau'] in X,
                     'finite_left_periphery': lambda X: X('finite') and ['T', 'C'] in X and 'T/prt' not in X,
                     'finite_tense': lambda X: 'T/fin' in X or (X('finite') and 'T' in X),
                     'preposition': lambda X: 'P' in X,
                     'expresses_concept': lambda X: ['N', 'Neg', 'P', 'D', 'φ', 'A', 'V', 'Adv', 'Q', 'Num', '0']
                     'SEM_internal_predicate': lambda X: 'SEM:internal' in X,
                     'SEM_external_predicate': lambda X: 'SEM:external' in X,
                     'scope_marker': lambda X: ['C', 'C/fin', 'OP'] in X,
                     'operator': lambda X: X.features(match=['OP:...']),
                     '-insitu': lambda X: X.features(match=['-insitu']),
                     'overt_phi': lambda X: 'ΦPF' in X,
                     'unrecognized_label': lambda X: ['CAT:?', '?'] in X,
                     'AgreeLF_occurred': lambda X: 'ΦLF' in X
                     }
```

The point is that we can define several linguistic notions in one place and then use notation `X.core('EPP')`, for example, to determine if X is an EPP-head. The phrase structure class contains an abstraction function which allows the user to substitute `X.core('EPP')` with `X('EPP')` such that the EPP-property is checked from the core of the head of X:

```
def __call__(self, stri):

    # Safeguard

    if not stri:
        return True

    # If the string stri provides a valid key for the dictionary containing the definition,
    # test whether the definition applies

    if abstraction_funct.get(stri):
        return abstraction_funct[stri](self)
```

The core has its own internal logic handling phi-feature/pro calculations and core-internal m-selection: anything that involves operations performed with lexical features. The following functions check m-selection properties between the feature bundles, modelling syntactic/semantic inflectional morphosyntax:

```
def integrity(self):

    # Examine each bundle and check m-selection

    for i in range(1, len(self._features)):
        if self.m_selection_violation(i):
            return True

def m_selection_violation(self, i):

    # Return a set of m-features of type [type]
    # Feature format is TYPE=A,B,C...

    def m_selection(fset, type):
        return set().union(*[set(f.split('=')[1].split(',')) for f in fset if f.startswith(type)])

    # --------------- MAIN FUNCTION ------------------------------------------------------

    # Check that the previous bundle (i-1) does not have negative m-selected features

    if self._features[i-1] & m_selection(self._features[i], '-mCOMP='):
        return True

    # Check that the previous bundle (i-1) contains positive m-selected features

    pos_mset = {f for f in m_selection(self._features[i], '+mCOMP=') if not f.endswith('/root')}
    pos_root = {f[:-5] for f in m_selection(self._features[i], '+mCOMP=') if f.endswith('/root')}
    if pos_mset:
        if not self._features[i-1] & pos_mset and not self.features() & pos_root:
            return True
```

Each feature bundle pair is examined for their m-selection features by the loop on lines 170-172. Both negative (186-187) and positive (191-195) m-selection features are checked. A feature which ends with **/root** is checked against all features of the item (line 194, second conjunct).

7.5 Semantics (narrow_semantics.py)

7.5.1 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.

```
def interpret_(self, X):
    if not X.copied:
        if X.zero_level():

            # Thematic roles

            if self.speaker_model.settings.retrieve('general_parameter_calculate_thematic_roles', True) and X('theta_predicate'):
                self.speaker_model.results.store_output_field('Thematic roles', self.thematic_roles_module.reconstruct(X))

            # Argument-predicate pairs

            if self.speaker_model.settings.retrieve('general_parameter_calculate_predicates', True) and \
                    'Φ' in X.core and not X('referential') and not X.INT({'N'}):
                self.speaker_model.results.store_output_field('Predicates', self.predicates.reconstruct(X))
                if self.speaker_model.settings.retrieve('UG_parameter_Agree', 'revised') == 'standard':
                    self.predicates.operation_failed = False
            if X.indexed_argument():
                self.speaker_model.results.store_output_field('Indexing by Agree', self.predicates.reconstruct_agreement(X))
            self.quantifiers_numerals_denotations.detect_phi_conflicts(X)
            self.interpret_tail_features(X)

            # Operator-variable constructions (scope reconstruction)

            if self.speaker_model.settings.retrieve('calculate_operator_bindings', True):
                self.speaker_model.results.store_output_field('Operator bindings', self.operator_variable_module.bind_operator(X))

            # Discourse features (pragmatic module)

            if self.speaker_model.settings.retrieve('calculate_DIS_features', True):
                self.speaker_model.results.store_output_field('Pragmatics', self.pragmatic_pathway.interpret_discourse_features(X))

            # Focus features (features with focus interpretation)

            if self.speaker_model.settings.retrieve('calculate_focus', True):
                self.speaker_model.results.store_output_field('Focus', self.focus.reconstruct(X))
            if self.failure():
                return
        else:
            self.interpret_(X.L())
            self.interpret_(X.R())
```

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

7.5.2 Projecting semantic inventories

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. 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)
```

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

7.5.3 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

---

[64] The assumption that every referential expression projects an entity into the discourse inventory might sound implausible, 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 projected, as shown by the code above, regardless of whether it will in the end be selected as a likely denotation

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.

7.5.4 Binding

Binding principles restrict assignment generation (§4.10.6). As an empirical phenomenon, binding describes coreference dependencies and limitations such as those illustrated in (70).

(70) a. John$_{a}$ admires himself$_{a,*b}$/him$_{*a,\ b}$/Bill$_{*a,b}$.

b. John$_{a}$ claimed that Bill$_{b}$ admires himself$_{*a,b}$/him$_{a,*b}$/John$_{*a,*b}$.

c. John's$_{a}$ brother$_{b}$ admires himself$_{*a,b}$/him$_{*a,*b}$/John$_{a,*b}$.

Because LPG handles these data by limiting assignment, it is claimed that the phenomenon emerges at the "outer edge" of language, at the language-cognition interface. Syntax internal computations do not get involved. The principles are implemented in the `SEM_quantifiers_numerals_denotations.py` module, a module handling semantic information processing that relates to quantifiable and countable "thing" objects such as persons. The relevant code is executed during postsyntactic semantic interpretation phase together with several other semantic calculations.

The overall information processing architecture is as follows. The postsyntactic semantic interpretation, activated by the speaker model (71)a-b and implemented by `SEM_narrow_semantics.py` module (71)c, takes complete LF interface representations as input and examines them recursively while reacting to lexical features (§7.5.1). Assignments (and denotations, binding) are relevant for *referential expressions* defined by lexical features such as [REF] (71)c-d.

(71) a. Deliver finished LF (Speaker model)

b. Activate postsyntactic semantic interpretation (Speaker model → Narrow semantics)

c. Activate assignment calculations (Narrow semantics → QND)

d. Create assignments and use binding (QND)

Narrow semantics is effectively a "gateway" which distributes the information acquired from the LF into semantic subsystems, assignment generation being one of them. The QND module computes regular referential dependencies that have to do with quantifiable and countable "thing" objects denoted by proper names, pronouns and other referential expressions. It will also contain calculations that have to do with pluralities and quantifiers. The semantic objects that will serve as the denotations of referential expressions are already present in the global discourse inventory, being generated by ontological projection before assignment calculations are attempted (by the narrow semantics module, function `inventory_projection`)(thus, project objects into ontology → assignments). The QND module associates each referential expression with possible denotations (a list of semantic objects in the global discourse inventory, as determined by their identifiers), which are listed in its narrow semantic entry. This process is restricted by lexical features such as phi-features. For example, if the global discourse inventory contains three singular male persons a, b and c, they will all occur as possible denotations for all referential expressions, such as male proper names, that have compatible phi-features (*John* ~ …{a, b, c}…). Possible denotations are used as a seed to calculate all assignments. The relevant code is inside `reconstruct_assignments` function in the QND module:

```
self.create_assignments_from_denotations(self.calculate_possible_denotations(X))
```

Thus, we calculate all possible denotations and use that information to calculate assignments. The function `calculate_possible_denotations` explores X recursively and assigns each referential expression linked with a narrow semantic object in the QND space (and hence also with a semantic object in the global inventory space) a set of possible denotations (e.g., *John* ~ {…a, b, c…}). The implementation code (excluding the line implementing recursion) is:

```
# If the head contains an index to QND-object, it will be considered
if X.core.has_idx('QND'):
    # Get the idx, space pair
    idx, space = X.core.get_idx_tuple('QND')
    # The field "Denotations" is used to store all denotations (objects in the global space)
    self.inventory[idx]['Denotations'] = self.create_all_denotations(X)
    log(f'\n\t\t{self.inventory[idx]["Reference"]}~{self.inventory[idx]["Denotations"]} ')
    # Return the list of denotations, which will be added together as the recursion proceeds
    referential_expressions_lst = [(idx, f'{X.illustrate()}', X, self.inventory[idx]['Denotations'])]
```

which executes if X is associated with a narrow semantic entry (first line). The denotations, provided as a list by the `create_all_denotations`, discussed below, are added to the dentations field in its narrow semantic entry in the QND space, which are all stored in `self.inventory` dictionary (thus, this refers to the inventory of the narrow semantic objects in the QND space). The referential expression together with the list of denotations is stored in a separate list (`referential_expressions_lst`) that will contain such information in connection with all referential expressions in X and which will be provided as input for assignment generation. Because assignment generation takes a list of referential expressions as input, it does not have access to anything else such as structure. Assignment generation is handed by the following code:

```
def create_assignments_from_denotations(self, referential_expressions_lst):
    #  Create all possible assignments (tup[3] = denotations)
    log(f'\n\tAssignments: ')
    for assignment in itertools.product(*[tup[3] for tup in referential_expressions_lst]):
        #   Create assignment dict
        assignment_dict = {tup[0]: assignment[i] for i, tup in enumerate(referential_expressions_lst)}
        #   Calculate assignment weights and add the (assignment, weight) pair into all_assignments
        self.all_assignments.append(self.calculate_assignment_weight(assignment_dict, referential_expressions_lst))
```

This function gets the `referential_expressions_lst` as input and creates the assignments, which are dictionaries, into `self.all_assignments` list. Assignments are generated by the iterator, which uses `itertools.product` returning all items in the Cartesian product of its argument, which in this case is a list of the denotations calculated for each referential expression (i.e. a list of lists). For example, [[a, b], [c, d]] (possible denotations for each expression) returns [a, c], [a, d], [b, c] and [b, d] (all combinations). Each assignment (`assignment_dict`) is a

dictionary which pairs expressions with denotation. The entries are generated by iterating over the `referential_expression_lst` and pairing expression indices ('name' for the expression) with the assignment:

```
assignment_dict = {tup[0]: assignment[i] for i, tup in enumerate(referential_expressions_lst)}
```

For example, if the assignments are [[a, b], [a, d]…] for two referential expressions *John* and *Bill*, we take each assignment and then create dictionary entries {*John*: a, *Bill*: b}, {*John*: a, *Bill*: d}… and add them to the assignment list after providing them with weights, which will become a list of assignment dictionaries. The keys cannot be the names (which may be identical after all), but unique indexes we get from `tup[0]`. See the generation of the referential expressions list above. Binding is part of the weight generation.

The function which creates all denotations for referential expressions returns a list of objects which are compatible with the expression as defined by its phi-features and the narrow semantic entry in the QND space (e.g., *John* can denote only male persons).

```
def create_all_denotations(self, X):
    return self.narrow_semantics.global_cognition.get_compatible_objects(
        self.inventory[X.head().core.get_referential_index('QND')])
```

As pointed above, compatibility checks that the semantic attributes associated with the expression ([GENDER:FEMININE]) match with properties of objects ('female') in the global discourse inventory, whatever these may be.

Binding calculations are part of the function (`calculate_assignment_weight`) which determines weights for assignments. Each expression is checked for binding principles violations (among other checks) in the context of each assignment provided by the caller:

```
189         for expression in ref_constituents_lst:
190
191             # Violations of binding conditions reduce the weight to 0
192
193             if not self.binding_conditions(expression, assignment):
194                 weighted_assignment['weight'] = 0
```

Binding conditions are checked for each binding-relevant feature (called “R-features”), from function `binding_conditions`:

```
for f in self.get_R_features(X):
    R, rule, intervention = self.parse_R_feature(f)

    # Check that no binding conditions are not violated
    # Function construct_semantic_working_memory will create a set of object based on at

    if binding_violation(assignment[idx],
                         rule,
                         X.construct_semantic_working_memory(intervention, assignment),
                         X):
        return False
```

The rest is a matter of the binding theory assumed in the background (Brattico 2025). Binding theory responds to *R-features* which form a two-dimensional system determining (i) how the assignment must be restricted in relation to what is in the semantic working memory and (ii) how the working memory is limited (locality). Component (i) is expressed by features NEW and OLD which determine whether the denotation must be new (=NEW) or old (=OLD) in relation to what is in the semantic working memory; component (ii) provides an intervention feature which limits the amount of nodes included in the working memory calculus. The features follow the format [R:A:B] where R provides the feature type, A is either NEW or OLD, and B is the intervention feature, if any. The features are provided by the lexical redundancy rules. Consider the Finnish first person singular pronoun *minä* ‘I’. Its lexical entry is

```
MINÄ        :: PF:minä LF:I N sg 1p def hum pron PHI_N LANG:FI
```

which provides that this word means ‘I’, it is first person (`1p`) singular (`sg`) human (`him`) pronoun (`pron`). The symbol `pron` triggers a lexical redundancy rule which provides the pronoun with the relevant R-features:

```
R_exp    :: iPHI:PRON:NONPRON     R:NEW
pron     :: iPHI:PRON:PRON        R:NEW:REF
refl     :: iPHI:PRON:PRON        R:OLD:REF
```

Thus, pronouns (marked as such by `pron`) will have feature [R:NEW:REF] which means that the denotation must be new in relation to the semantic working memory limited by feature [REF].

This captures the profile of pronouns illustrated in (70).[65] The above entries also show the corresponding R-features for regular referential expressions and reflexives. This system is enforced by the `binding_violation` function:

```
def binding_violation(semantic_object, rule_, semantic_working_memory, X):

    # Semantic working memory (whether complete, limited) is constructed by the caller and
    # is here provided as a set

    # RULE 1
    # If the same semantic object denoted by X is inside (complete, limited) semantic working memory but
    # X marked as NEW, raise violation

    if 'NEW' in rule_ and {semantic_object} & semantic_working_memory:
        log(f'-Illegitimate binder for {X.max().illustrate()}({semantic_object}) in WM ')
        return True

    # RULE 2
    # If the same semantic object denoted by X is not inside (complete, limited) semantic working memory
    # but X is marked for OLD, raise violation

    if 'OLD' in rule_ and not {semantic_object} & semantic_working_memory:
        log(f'-Binder missing for {X.max().illustrate()}({semantic_object}) from WM ({semantic_working_memory})')
        return True
```

The code is self-explanatory due to the comments. The relevant semantic working memory is constructed by the caller and takes the intervention feature into account. It is generated by relying on external search (§4.6):

```
def construct_semantic_working_memory(X, intervention_feature, assignment):
    sWM = X.EXT(acquire='all', criteria=lambda x: x.head().core.get_idx_tuple('QND') and x.head() != X and not x.copied)
    limit = next((i for i, x in enumerate(sWM) if intervention_feature in x.head().core), len(sWM)) + 1
    return {assignment[x.head().core.get_referential_index('QND')] for x in sWM[0:limit]}
```

The first line returns all relevant referential items in the external search path; the second line sets the limit on the first item that has the intervention feature; the third line returns a set of referential indices of the chosen items.[66]

---

[65] The empirical consequence is that a sentence such as *John admires him* cannot mean 'John admires himself', since under every assignment the denotation of *him* must be new in relation to the denotation of *John*. Because the semantic working memory is limited by [REF], *him* can still denote John in *John claims that Mary admires him*.

[66] The external search function EXT has its own intervention argument, but it cannot be used here since, contrary to what is required here, EXT does not return the intervening element but only uses it as a halting criterion. Binding calculations presuppose that the intervening element is included in the set of returned elements. This is an obvious issue that needs attention in the future.

7.5.5 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:

```
def bind_operator(self, X):
    self.bindings = []

    # Scope markers do not themselves have scopes

    if not X('scope_marker'):

        # Find all operator features

        for Opf in (f for f in X.core.features() if self.is_operator_feature(f)):

            # Determine the scope for the operator and, if not found, determine covert scope

            binding = self.interpret_covert_scope(self.find_overt_scope(X, Opf), Opf)

            # Produce output

            self.interpret_results(binding, Opf)

    return self.bindings
```

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

```
def find_overt_scope(X, Opf):

    # Scope information is returned in the form of a dictionary

    return next(({'Head': X, 'Scope': Y, 'Overt': True} for Y in X.EXT(acquire='all') if
                 {'OP', 'Fin', Opf} in Y.core), {'Head': X, '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_dict, Opf):

    # Determine covert scope if no scope exists and in situ is licensed

    if not binding_dict['Scope'] and '-insitu' not in binding_dict['Head'].core.bundle_features(Opf):

        # Covert scope markers are elements of the finite left periphery, if not found, return Scope: None

        return next(({'Head': binding_dict['Head'], 'Scope': X, 'Overt': False}
                     for X in binding_dict['Head'].EXT(acquire='all', self=True) if
                     X('finite_left_periphery')),
                    {'Head': binding_dict['Head'], 'Scope': None, 'Overt': False})

    return binding_dict
```

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

7.5.6 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.

Most of the operations of this function, which were created and discussed for (Brattico, 2021a), have been disabled for this version (20.1), since it become clear that a rethinking might be necessary. Specifically, while the discourse features are still active, the way the topic and focus gradient are determined seems to require rethinking. I will nevertheless leave a description of the old code here, since it is still present.

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. 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.

```
def calculate_information_structure(self, ps):
    log('\n\t\tCalculating information structure...')
    return self.create_topic_gradient(self.collect_arguments(ps))
```

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:

```
def allocate_attention(self, head):
    if head.core('referential') or head.core('preposition'):
        idx = self.consume_index()
        head.core.add_features({'*IDX:' + str(idx)})
        self.records_of_attentional_processing[str(idx)] = {'Constituent': head.max(), '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.[67]

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.

```
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}
```

[67] 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.

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.

7.5.7 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 search 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.

```
def identify_argument(X):
    arguments = [acquire(X) for acquire in [lambda x: x.generate_pro(),
                                            lambda x: x.complement() and x.complement().INT('referential'),
                                            lambda x: x.indexed_argument(),
                                            lambda x: x.specifier(),
                                            lambda x: x.control()]]
    return next((x for x in arguments if x), None)
```

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 the figure below.

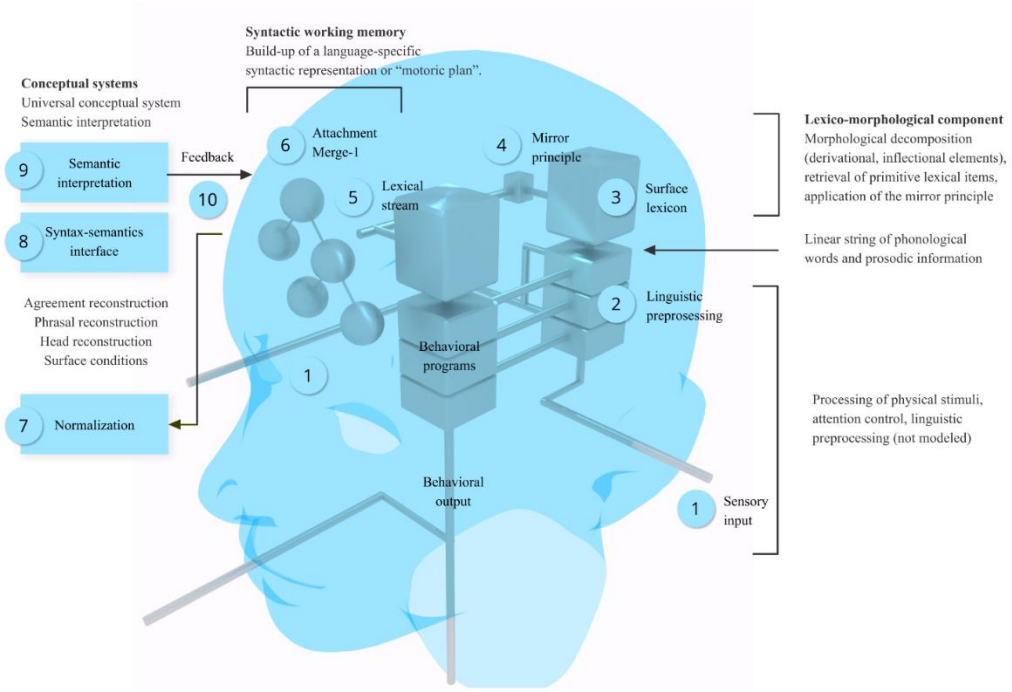


Fig. The speaker model contains all submodules defined by the theory and their connections.

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 `derivational_search_function`.

8.1.3 Derivational search function

The function accepts keyword arguments as follows:

```
78      def derivational_search_function(self, **kwargs):
79          X = kwargs.get('phrase_structure', None)
80          lst = kwargs.get('word_list', [])
81          index = kwargs.get('index', 0)
82          infl_buffer = kwargs.get('infl_buffer', [set()])
83          prosody = kwargs.get('prosody', set())
```

`X` = current phrase structure in the syntactic working memory; `lst` = the input sentence as a list of phonological words; `index` = the position in the sentence we are processing/attending currently; `infl_buffer` = inflectional features that have been consumed and inserted into a

short-term memory buffer; `prosody` = any prosodic features currently in a short-term memory. The short-term memory buffers are required when processing stacked inflectional features which must all be inserted inside heads.

The function 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:

```
186     def circuit_breaker(self, X, lst, index):
187         set_logging(True)
188         if self.exit:
189             return True
190 
191         # If there are no more words in the inpu
192 
193         if index == len(lst):
194             self.evaluate_complete_solution(X)
195             return True
```

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.

```
# Retrieve lexical items on the basis of phonological input

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.

```
# Examine all retrieved lexical items (ambiguity resolution)

for lex 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.

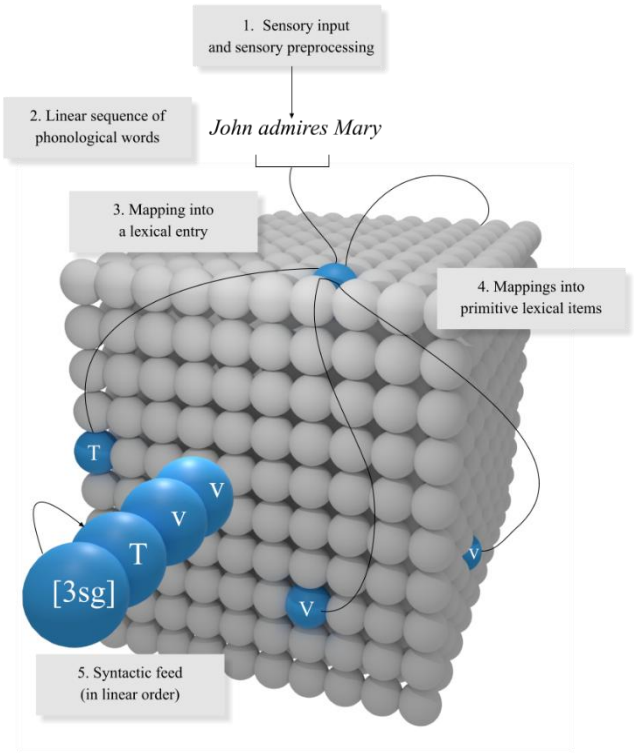


Fig. Organization of the lexicon. The lexicon is a mapping from phonological surface strings into constituents which make up the lexicon. Some surface strings map into primitive lexical items (blue items) which are sets of lexical features (e.g., /the/ ~ D). Other surface strings map into morphological 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. 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). The *first item* in the new list will be sent to the syntactic component via lexical stream. 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.

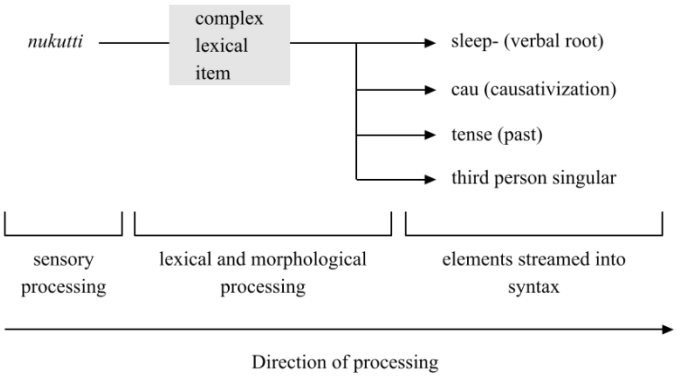


Figure 41. Lexical processing pipeline.

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.

```
for N in self.plausibility_metrics.filter_and_rank(X, W):

    # Create candidate solution [[X...] W]

    Y = X.target_left_branch_and_copy(N).transfer().attach(W.copy())
    log(f'\n\t= {Y.top()}\n')
    PhraseStructure.cyclic = True
    log(f'\n\tCyclic reconstruction:\n')

    # Cyclic reconstruction

    Y = Y.bottom().reconstruct()

    # Consume next item (recursion)

    self.derivational_search_function(phrase_structure=secure_copy(Y), word_list=lst, index=index + 1)
```

Each site from the ranking is then explored. 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. 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_copy` 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 depending on the situation and performs working memory operations.

```
def attach(X, W):
    X.speaker_model.results.consume_resources('Merge', X, W)
    if X.w_internal() or X.INT({'C'}) or X.bottom_affix().INT('EHM'):
        return X ** W   #   EHM
    return X * W        #   Phrasal Merge (asymmetric)
```

Once all words have been processed, the result will be submitted to a finalization stage implemented by function `evaluate_complete_solution`. This process will apply transfer, LF-legibility and semantic interpretation. 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. The parser will get a filtered and ranked list in return, which is used by the parser to explore solutions.

```
def filter_and_rank(self, X, w):
    if X.bottom().w_internal():
        return [X.bottom()]
    return self.rank(self.filter(X.collect_sWM(geometrical=True, self=True)), w)
```

If the incoming word was part of the previous word, it will always be merged as its sister. We will therefore only return one solution. Otherwise, filter and ranking will be applied, in that order. 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.

```
def filter(self, X_right_edge):
    return [N for N in X_right_edge if N.zero_level() or self.left_branch_filter(N)]
```

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.

```
def left_branch_filter(self, X):
    set_logging(False)
    self.speaker_model.LF.active_test_battery = self.left_branch_filter_test_battery
    return self.speaker_model.LF.pass_LF_legibility(X.copy().transfer(), logging=False)
```

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().

```
def create_baseline_weighting(self, weighted_site_list, method=''):
    if method == 'Z':
        new_weighted_site_list = [(site, j) for j, (site, w) in enumerate(weighted_site_list, start=1)]
        new_weighted_site_list[0] = (new_weighted_site_list[0][0], len(weighted_site_list))
        new_weighted_site_list[-1] = (new_weighted_site_list[-1][0], len(weighted_site_list)+1)
        new_weighted_site_list = [(site, weight-1) for site, weight in new_weighted_site_list]
        return new_weighted_site_list
    if method == 'Random':
        site_w = list(range(len(weighted_site_list)))
        random.shuffle(site_w)
        return [(site, site_w[i]) for i, (site, w) in enumerate(weighted_site_list, start=0)]
    if method == 'Top-down':
        return [(site, -j) for j, (site, w) in enumerate(weighted_site_list, start=1)]
    if method == 'Bottom-up':
        return [(site, j) for j, (site, w) in enumerate(weighted_site_list, start=1)]
    if method == 'Sling':
        lst = [(site, -j) for j, (site, w) in enumerate(weighted_site_list, start=1)]    # Top-down
        lst[-1] = (lst[-1][0], 1)                                                        # Promote the bottom node
        return lst
    else:
        return [(site, j) for j, (site, w) in enumerate(weighted_site_list, start=1)]
```

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. 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.

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.

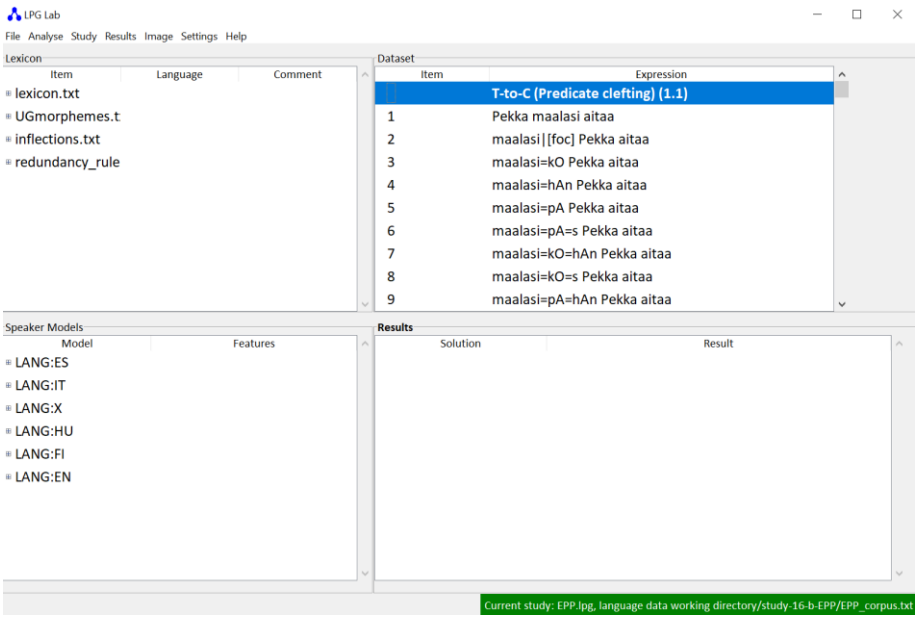
Resource counting is implemented by the parser and is recorded into a dictionary with keys referring to the type of operation (e.g., *Merge*, *Move Head*), value to the number of operations before the first solution was found.

```
self.resources = {"Total Time": {'ms': 0, 'n': 1},
                  "Garden Paths": {'ms': 1458, 'n': 0},
                  "Sensory Processing": {'ms': 75, 'n': 0},
                  "Lexical Retrieval": {'ms': 50, 'n': 0},
                  "Merge": {'ms': 7, 'n': 0},
                  "Head Chain": {'ms': 7, 'n': 0},
                  "Phrasal Chain": {'ms': 7, 'n': 0},
                  "Feature Inheritance": {'ms': 7, 'n': 0},
                  "Agree": {'ms': 7, 'n': 0},
                  "Feature": {'ms': 7, 'n': 0},
                  "Left Scrambling": {'ms': 7, 'n': 0},
                  "Right Scrambling": {'ms': 7, 'n': 0},
                  "Extraposition": {'ms': 7, 'n': 0},
                  "Last Resort Extraposition": {'ms': 7, 'n': 0},
                  "Mean time per word": {'ms': 0, 'n': 1}
                  }
```

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.

9 Graphical user interface

The system can be used via a separate graphical user interface (GUI).[68] The interface can be launched by command `python lpparse -app` in the command prompt. Instead of running one study, the command opens up the following interface:



The program will first attempt to locate the application configuration file `$app_settings.txt` from the root directory and, if it is found, reads the name and folder of the study configuration

---

[68] The interface was originally designed to replace the previous phrase structure image drawing algorithm that relied on the pyglet library that was not part of the standard Python installation. In addition, the current image generation component is based on tkinter, a library that can be used to design proper graphical user interfaces and therefore also more sophisticated drawing tools. The tree drawing algorithm in turn was originally designed to illustrate the output of the algorithm (it replicates what is in the output and does not allow the user to add or remove anything) and still serves this function, however as the graphical interface component expanded in scope it now offers several additional tools for the examination of the output of the model. This is what this interface is still primarily designed for, thus as it stands now it is not yet possible to replace all work with the command line tools with this interface although this may be an option in the future.

file (`*.lpg`). The interface is then populated by the data and other properties as defined in the study configuration file. In addition to inspecting the results of a study visually, the program can be used to draw standardized phrase structure images from scratch. There is a separate manual (`LPGlab_manual.pdf`) in the `/doc` folder, thus its properties are not reviewed here.

References

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). Computation and the justification of grammars. *Finno-Ugric Languages and Linguistics*, *10*(1–2), 51–74.

Brattico, P. (2021c). Computational linguistics as natural science. *Lingbuzz/005796*. https://ling.auf.net/lingbuzz/005796?_s=MUzSsZPWMdmoX8nI&_k=3e1ty8tPIVI3hVmY

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

Brattico, P. (2022). 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. (2023a). Computational analysis of Finnish nonfinite clauses. *Nordic Journal of Linguistics*, *X*(X), X–X.

Brattico, P. (2023b). Computational analysis of Finnish nonfinite clauses. *Nordic Journal of Linguistics*, 1–40. https://doi.org/10.1017/S0332586523000082

Brattico, P. (2023c). Structural case assignment, thematic roles and information structure. *Studia Linguistica*, *77*(1), 172–217. https://doi.org/10.1111/stul.12206

Brattico, P. (2024). Computational generative grammar and complexity. *Software Documentation for Python Scripts Implementing Computational Generative Grammars*. https://github.com/pajubrat/Templates

Brattico, P. (2025). Binding in Finnish and the language-cognition interface. *Journal of Uralic Languages*, *x*, x–x.

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

Chomsky, N. (1959). Review of `"Verbal Behaviour". *Language*, *35*(1), 26–58.

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. (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.

Chomsky, N. (2013). Problems of projection. *Lingua*, *130*, 33–49.

Chomsky, N. (2015). Problems of projection: Extensions. In E. Di Domenico, C. Hamann, & S. Matteini (Eds.), *Structures, Strategies and Beyond: Studies in honour of Adriana Belletti* (pp. 1–16). https://doi.org/10.1075/la.223.01cho

Chomsky, N. (2021). Minimalism:Where Are We Now, and Where Can We Hope to Go. *Gengo Kenkyu*, *160*, 1–41.

Chomsky, N., Gallego, Á. J., & Ott, D. (2019). Generative Grammar and the Faculty of Language: Insights, Questions, and Challenges. *Catalan Journal of Linguistics*, *Special Is*, 229–261.

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

Huang, C.-T. J. (1982). *Logical relations in Chinese and the theory of grammar*. Garland.

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

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

Kayne, R. (1994). *The Antisymmetry of Syntax*. MIT Press.

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

Marcolli, M., Berwick, R. C., & Chomsky, N. (2023). Old and New Minimalism: a Hopf algebra comparison. *ArXiv:2306.10270*.

Marcolli, M., Chomsky, N., & Berwick, R. (2023). Mathematical Structure of Syntactic Merge. *ArXiv:2305.18278*.

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*.

Salo, P. (2003). *Causative and the Empty Lexicon: A Minimalist Perspective*.

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