

Computational implementation of a linear phase parser. Framework and technical
documentation
(version 12.0)

2019

(Revised May 2022)

Pauli Brattico

Abstract

This document describes a computational implementation of a linear phase parser. The model assumes that the core computational operations of narrow syntax are applied incrementally on a phase-by-phase basis in language comprehension. Full formalization with a description of the Python implementation will be discussed, together with a few words towards empirical justification. The theory is assumed to be part of the human language faculty (UG).

How to cite this document: Brattico, P. (2019). Computational implementation of a linear phase parser. Framework and technical documentation (version 12.0). IUSS, Pavia.

1	INTRODUCTION	6
2	INSTALLATION AND BASIC USE	7
2.1	INSTALLATION	7
2.2	PROGRAM FILES AND FOLDER STRUCTURE	8
2.3	USE.....	9
2.4	STRUCTURE OF THE SCRIPT	12
2.5	REPLICATION.....	14
2.6	DEFAULT TEST CORPUS.....	14
3	FRAMEWORK	16
3.1	THE FRAMEWORK.....	16
3.2	COMPUTATIONAL LINGUISTICS METHODOLOGY	20
3.3	OVERVIEW OF THE HYPOTHESIS.....	21
4	THE KERNEL (COMPREHENSION CYCLE)	27
4.1	INTRODUCTION.....	27
4.2	MERGE-1.....	27
4.3	LEXICAL SELECTION FEATURES	31
4.4	PHASES AND LEFT BRANCHES	33
4.5	LABELING	33
4.6	UPWARD PATHS AND MEMORY SCANNING	35
4.7	ADJUNCT ATTACHMENT.....	36
4.8	TRANSFER.....	39
4.8.1	<i>Introduction</i>	<i>39</i>
4.8.2	<i>Phrasal reconstruction: Some general comments</i>	<i>39</i>
4.8.3	<i>Ā-chains</i>	<i>41</i>
4.8.4	<i>A-chains and EPP</i>	<i>44</i>
4.8.5	<i>Head reconstruction</i>	<i>45</i>
4.8.6	<i>Adjunct reconstruction</i>	<i>47</i>
4.8.7	<i>Minimal search</i>	<i>48</i>
4.8.8	<i>Agree-1</i>	<i>49</i>
4.8.9	<i>Ordering of operations</i>	<i>52</i>
4.9	LEXICON AND MORPHOLOGY	52
4.9.1	<i>From phonology to syntax</i>	<i>52</i>
4.9.2	<i>Lexical items and lexical redundancy rules</i>	<i>53</i>
4.9.3	<i>Derivational and inflectional morphemes</i>	<i>53</i>

4.9.4	<i>Morphology</i>	54
4.10	NARROW SEMANTICS.....	54
4.10.1	<i>Syntax, semantics and cognition: the framework</i>	54
4.10.2	<i>Incremental semantics</i>	57
4.10.3	<i>Argument structure</i>	58
4.10.4	<i>Antecedents and control</i>	60
4.10.5	<i>The pragmatic pathway</i>	61
4.10.6	<i>Operators</i>	61
4.10.7	<i>Binding</i>	62
5	PERFORMANCE.....	64
5.1	HUMAN AND ENGINEERING PARSERS.....	64
5.2	MAPPING BETWEEN THE ALGORITHM AND BRAIN	64
5.3	COGNITIVE PARSING PRINCIPLES	65
5.3.1	<i>Incrementality</i>	65
5.3.2	<i>Connectness</i>	70
5.3.3	<i>Seriality</i>	71
5.3.4	<i>Locality preference</i>	71
5.3.5	<i>Lexical anticipation</i>	72
5.3.6	<i>Left branch filter</i>	72
5.3.7	<i>Working memory</i>	72
5.3.8	<i>Conflict resolution and weighting</i>	73
5.4	MEASURING PREDICTED COGNITIVE COST OF PROCESSING	73
5.5	A NOTE ON IMPLEMENTATION	74
6	INPUTS AND OUTPUTS	75
6.1	INSTALLATION AND USE	75
6.2	GENERAL ORGANIZATION.....	77
6.3	STRUCTURE OF THE INPUT FILES	81
6.3.1	<i>Test corpus file (any name)</i>	81
6.3.2	<i>Lexical files (lexicon.txt, ug_morphemes.txt, redundancy_rules.txt)</i>	82
6.3.3	<i>Lexical redundancy rules</i>	84
6.3.4	<i>Study parameters (config_study.txt)</i>	85
6.4	STRUCTURE OF THE OUTPUT FILES.....	85
6.4.1	<i>Results</i>	85
6.4.2	<i>The log file</i>	86
6.4.3	<i>Simple logging</i>	89
6.4.4	<i>Saved vocabulary</i>	89
6.4.5	<i>Images of the phrase structure trees</i>	89

6.4.6	<i>Resources file</i>	90
6.4.7	<i>Semantic interpretation</i>	91
6.5	MAIN SCRIPT	91
7	GRAMMAR FORMALIZATION	93
7.1	BASIC GRAMMATICAL NOTIONS (PHRASE_STRUCTURE.PY)	93
7.1.1	<i>Introduction</i>	93
7.1.2	<i>Types of phrase structure constituents</i>	93
7.1.2.1	Lexical items	93
7.1.2.2	Primitive and terminal constituents	93
7.1.2.3	Complex constituents; left and right daughters	94
7.1.2.4	Complex heads and affixes	94
7.1.2.5	Sisters	95
7.1.2.6	Complement, proper complement	95
7.1.2.7	Labels	96
7.1.2.8	Minimal search, geometrical minimal search and upstream search	96
7.1.3	<i>Upward paths</i>	97
7.1.4	<i>Basic structure building</i>	98
7.1.4.1	Cyclic Merge	98
7.1.4.2	Countercyclic Merge-1	99
7.1.4.3	Remove	100
7.1.4.4	Detachment	100
7.1.5	<i>Nonlocal grammatical dependencies</i>	100
7.1.5.1	Probe-goal: probe(label, goal_feature)	100
7.1.5.2	Tail-head relations	101
7.2	TRANSFER (TRANSFER.PY)	103
7.2.1	<i>Introduction</i>	103
7.2.2	<i>Head reconstruction (head_reconstruction.py)</i>	103
7.2.3	<i>Adjunct reconstruction (adjunct_reconstruction.py)</i>	106
7.2.4	<i>Ā-reconstruction (phrasal_reconstruction.py)</i>	108
7.2.5	<i>A-reconstruction</i>	110
7.2.6	<i>Extraposition as a last resort (extraposition.py)</i>	110
7.2.7	<i>Adjunct promotion (adjunct_constructor.py)</i>	111
7.3	AGREEMENT RECONSTRUCTION AGREE-1 (AGREEMENT_RECONSTRUCTION.PY)	112
7.4	LF-LEGIBILITY (LF.PY)	114
7.5	SEMANTICS (NARROW_SEMANTICS.PY)	115
7.5.1	<i>Introduction</i>	115
7.5.2	<i>Projecting semantic inventories (semantic switchboard)</i>	116
7.5.3	<i>Quantifiers-numerals-denotations module</i>	117
7.5.4	<i>Operator-variable interpretation (SEM_operators_variables.py)</i>	121

7.5.5	<i>Pragmatic pathway</i>	122
7.5.6	<i>LF-recovery</i>	124
8	FORMALIZATION OF THE PARSER	127
8.1	LINEAR PHASE PARSER (LINEAR_PHASE_PARSER.PY).....	127
8.1.1	<i>The brain model</i>	127
8.1.2	<i>Recursive parsing function (_first_pass_parse)</i>	128
8.2	PSYCHOLINGUISTIC PLAUSIBILITY	133
8.2.1	<i>General architecture</i>	133
8.2.2	<i>Filtering</i>	134
8.2.3	<i>Ranking (rank_merge_right_)</i>	134
8.2.4	<i>Knocking out heuristic principles</i>	136
8.3	RESOURCE CONSUMPTION	136

1 Introduction

This document describes a computational Python based implementation of a linear phase parser that was originally developed and written by the author while working in an IUSS-funded research project between 2018-2020, in Pavia, Italy, and then continued as an independent project.¹ The algorithm is meant as a realistic description of the information processing steps involved in real-time language comprehension. It captures both cognitive principle of language (“competence”) and cognitive mechanisms involved in language comprehension and use (“performance”).

This document describes properties of the version 12.0, which keeps within the framework of the original work but provides improvements, corrections and additions. This document does not, however, substitute for a proper scientific treatment of the topics covered. There are very few references, and the topics are organized into a form of a tutorial that illustrates the core issues by using the simplest possible examples imaginable. The material is targeted for researchers and computer scientists who need an entry point for understanding the source code and/or for working out a similar system on their own. Proper scientific discussion can be found from the published articles and especially from technical supplementary appendices linked to those articles.²

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

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

² The model has been applied to filler-gap dependencies, \bar{A} -reconstruction and pied-piping, local and nonlocal (\bar{A}) head movement, word order and adjunction, control and agreement, case assignment, information structure, binding, clitics, and syntactic working memory.

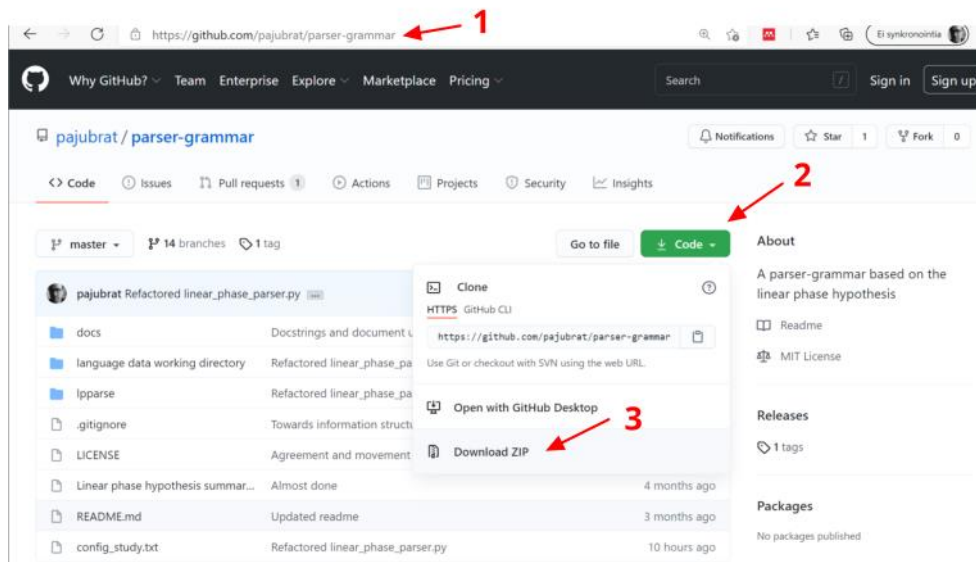
2 Installation and basic use

2.1 Installation

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

<https://github.com/pajubrat/parser-grammar>

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



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

Because the script is written in Python (3x) programming language, you will need Python on your local machine. Python is an interpreter that reads the script (which exists as ordinary text

files) and translates it into concrete machine operations. If you do not have Python installed, navigate to the following web address and follow the instructions there:

<https://www.python.org/>

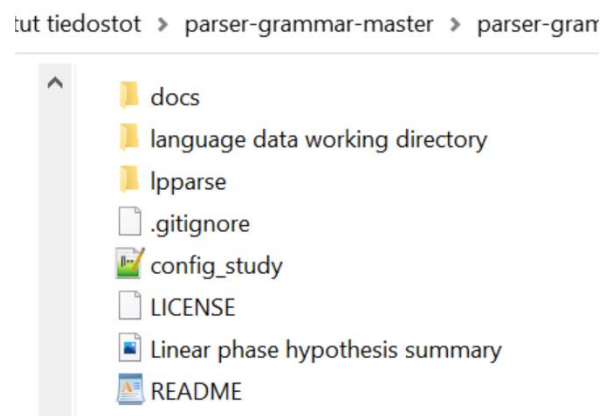
You can install Python into any directory.

Finally, in Windows you will have to include the Python installation folder to Windows PATH environmental variable so that Windows can find the Python interpreter when you execute the script. Search for the term “setting windows PATH variables” for instructions on how to do this on your system. The PATH variable must list the folder that contains your Python installation.

Some versions and/or uses of the program rely on external Python libraries which provide auxiliary functionality, such as phrase structure tree images and numerical analyses (pandas, matplotlib, numpy). The linguistic analysis component relies on Python 3.x only and does not require any external modules. If the program does not execute properly but asks for external libraries, then the user should either install them (search “install Python libraries”) or the program should be run in a bare mode that does not require any external functionality. To use the program in a bare mode, disable image drawing (by editing file `config_study.txt`, line `datatake_images: False`) and remove (or comment out) all reference to pandas, matplotlib and numpy libraries in the module `__main__.py`.

2.2 Program files and folder structure

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



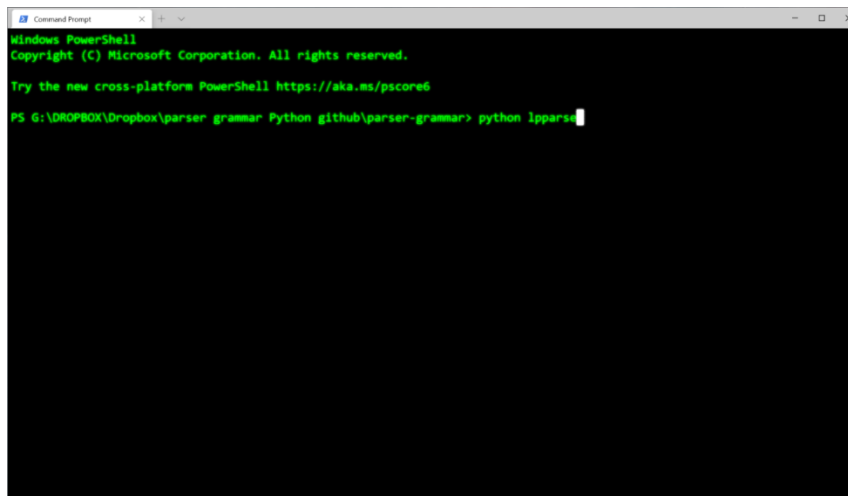
Folder `/docs` contains documentation (e.g., this document), `/language data working directory` stores the input and output files associated with any particular study, and `/lpparse` stores the modules containing the program code. The file `config_study.txt` is a text file that is used to

parametrize the operation of the script, in a manner explained below. Notice that the contents shown in the above Figure may change as the source code is developed.

2.3 Use

The script is launched by having the Python interpreter to run the program script. In most cases the version you copied or cloned from the code repository comes in a “working configuration,” meaning that it should do something meaningful out of the box. It is important to understand, however, that the version you 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.

To launch the script in Windows, start a command prompt program such as Windows PowerShell or the command prompt (“cmd”) and then, when you are in the command prompt, navigate to the local installation directory (or launch the command prompt directly in the installation folder). The script can be executed by command *python lpparse*, which is written into the command shell as shown below. This asks the Python interpreter to run the main program module from the folder *lpparse*. I will return to the organization of the program modules below.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS G:\DROPBOX\parser grammar Python github\parser-grammar> python lpparse
```

The program first reads a study configuration file *config_study.txt* from the local installation directory, which contains the parameters used in the simulation trial. This file is a “master control” utility that will tell the script what to do. One of these parameters is the folder and name of the *test corpus file* that contains the sentences the script will analyze. The user can open the study configuration file by using any text editor (such as Notepad). Here is a screenshot of the file as it exists currently on my local installation directory:

```

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

```

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

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

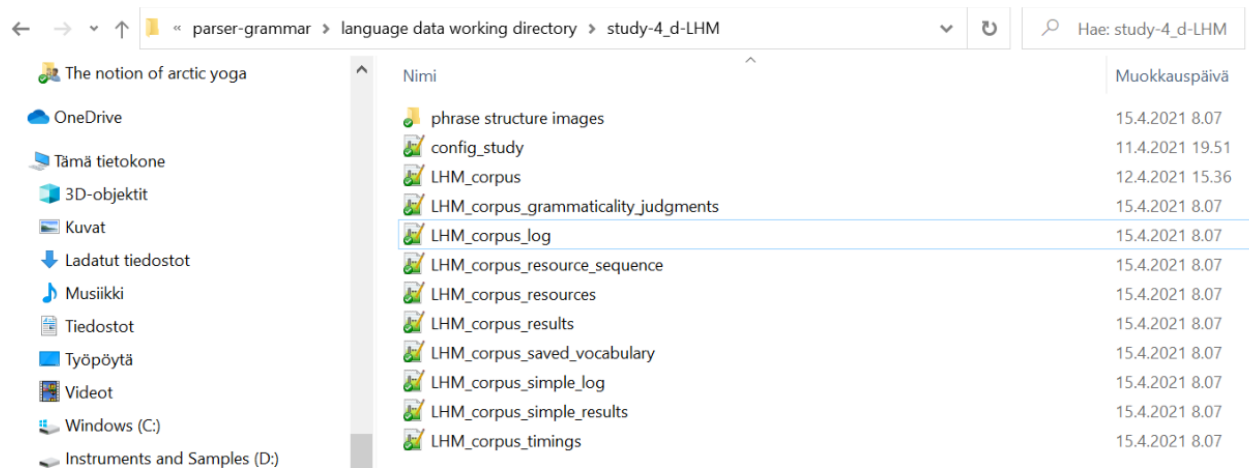
```

221 taytyy#[pA]#[hAn] Pekan_gen ihailla Merjaa
222 taytyy#[foc]#[kO]#[hAn] Pekan_gen ihailla Merjaa
223 taytyy#[foc]#[pA]#[hAn] Pekan_gen ihailla Merjaa
224
225 taytyy#C/op Pekan_gen myyda omaisuuttaan
226
227 & Group 2.1.4 Want-to-C
228
229 haluaa#[foc] Pekka ihailla Merjaa
230 haluaa#[hAn] Pekka ihailla Merjaa
231 haluaa#[kO] Pekka ihailla Merjaa
232 haluaa#[pA] Pekka ihailla Merjaa
233 haluaa#[foc]#[hAn] Pekka ihailla Merjaa
234 haluaa#[foc]#[kO] Pekka ihailla Merjaa
235 haluaa#[foc]#[pA] Pekka ihailla Merjaa
236 haluaa#[kO]#[hAn] Pekka ihailla Merjaa
237 haluaa#[pA]#[hAn] Pekka ihailla Merjaa
238 haluaa#[foc]#[kO]#[hAn] Pekka ihailla Merjaa
239 haluaa#[foc]#[pA]#[hAn] Pekka ihailla Merjaa
240
241 haluaa#C/op Pekka myyda omaisuuttaan
242
243 & Group 2.1.5 Aux-to-C
244
245 on'#C/op Pekka ihaillut Merjaa
246 on'#[foc] Pekka ihaillut Merjaa
247 on'#[hAn] Pekka ihaillut Merjaa
248 on'#[kO] Pekka ihaillut Merjaa
249 on'#[pA] Pekka ihaillut Merjaa
250 on'#[foc]#[hAn] Pekka ihaillut Merjaa
251 on'#[foc]#[kO] Pekka ihaillut Merjaa
252 on'#[foc]#[pA] Pekka ihaillut Merjaa
253 on'#[kO]#[hAn] Pekka ihaillut Merjaa
254 on'#[pA]#[hAn] Pekka ihaillut Merjaa
255 on'#[foc]#[kO]#[hAn] Pekka ihaillut Merjaa
256 on'#[foc]#[pA]#[hAn] Pekka ihaillut Merjaa
257
258 on'#C/op Pekka myynyt omaisuuttaan
259
260 & Group 2.1.6 All constructions (2.1.1-2.1.5) with formal C-feature C/fin
261
262 ihailee#C/fin Pekka Merjaa
263 ei#C/fin Pekka ihaile Merjaa
264 taytyy#C/fin Pekan_gen ihailla Merjaa
265 haluaa#C/fin Pekka ihailla Merjaa
266 on'#C/fin Pekka ihaillut Merjaa

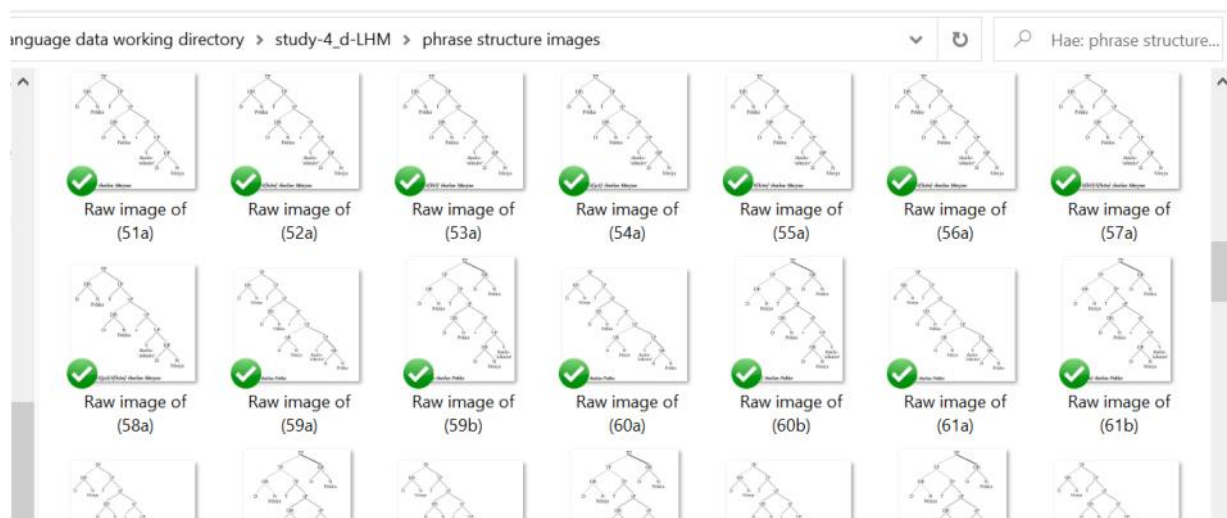
```

This is a screenshot of the list of sentences that the script will process when it is executed by writing *python lpparse* into the command prompt. If the user wants to use another file, say test sentences in some other language, the name and folder should be changed accordingly in the configuration file.

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



The first file is the test corpus, followed by several outputs generated by the algorithm. For example, the file `LHM_corpus_grammaticality_judgments.txt` contains a list of the original sentences and the grammaticality judgments provided by the script. These files can be opened by any text editor. You can also see a folder `/phrase structure images` at the top of the directory. If you activated image generation in the configuration file, then this folder will contain a phrase structure image for each grammatical input sentence, as provided by the model:

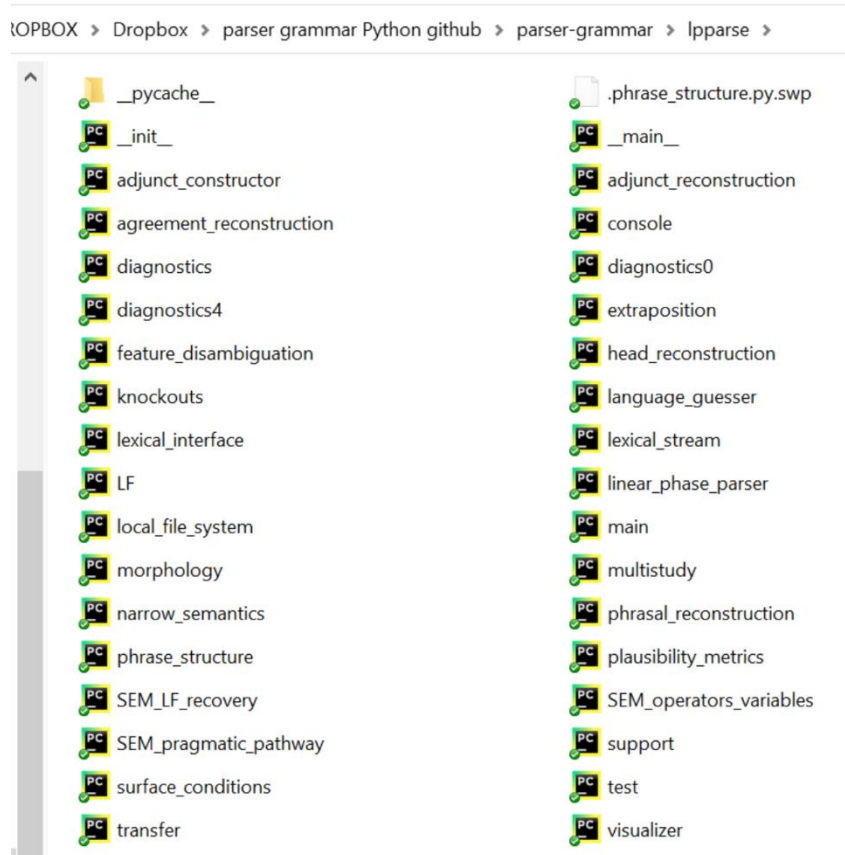


These images, which are raw data produced by the algorithm and replicas of the text output, are useful when the analysis returned by the script is complex or if the user wants to eyeball the results quickly. They could be used in printed research articles. In order to generate these images, the user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

2.4 Structure of the script

When the user runs the script by typing `python lpparse` inside the installation director, what happens under the hood is that the Python interpreter will run a main script (called `__main__.py`)

from the folder /lpparse. This folder contains all the program files and modules which define the theory and the behavior of the whole system:

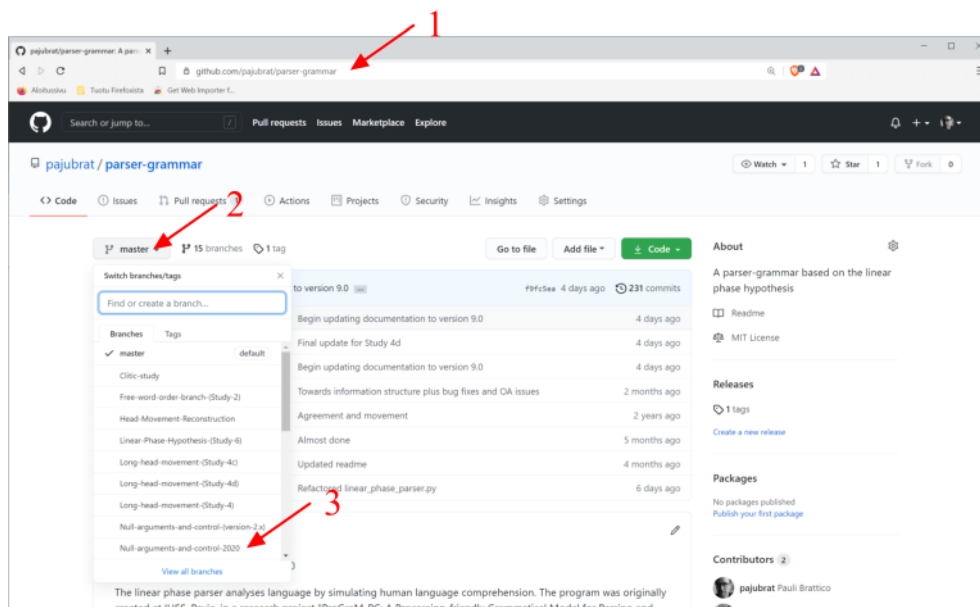


The `__main__.py` module will take control of the execution and call the program code that exists inside the individual files. These files (also called modules) correspond closely to the contents of the empirical theory that they implement. For example, the module `narrow_semantics.py` contains operations that correspond to an empirical component carrying the same name in the theory. These files can be opened and edited by any text editor. Thus, if the researcher wants to find out exactly how some grammatical operation, principle or module operates, this information can be found from these files. For example, the underlying notion of phrase structure is defined in the module `phrase_structure.py`. If the user changes any of these files and runs the script anew, then the modified script will be run, and the original output files will be overwritten. Python is an interpreted language, meaning that there is no separate executable; each time the user runs the script these text files are consulted.³

³ This is not literally true. What is important is that any functional change into the text files will automatically change the behavior of the system.

2.5 Replication

When a study is published, the input files plus its source code is stored somewhere. One such place is the source code repository itself. The source code repository (that runs on a program called Git) not only stores the current version of the script, but also maintains a record of previous versions. The user can therefore backtrack to a previous point in the development. The development history can also be branched, meaning that it is possible to develop several versions of the software in parallel (and possible merge them later). These parallel branches are currently used to store unambiguous snapshots of the scripts that were used in connection with published studies. To access them, navigate again to the source code repository (1, Figure below), then click for the tab shown in the figure below (2) and select the branch that is of interest (3).



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

2.6 Default test corpus

A single linguistic study will usually focus on one narrow phenomenon in the interest of systematic and comprehensive coverage. This means that each such study is usually associated with a test corpus that contains a very specific set of test sentences, exhibiting some linguistic phenomenon and little else. This raises the question of what happens when the model is developed to incorporate new datasets. One possibility is that the model is revised so radically that it fails to

handle previous datasets. In such case, the model development was not cumulative; instead it regressed by undoing previous results. Sometimes this is desired, but ultimately want to create a model that captures as much data as possible. To make cumulative development possible, a default test corpus was created which contains the core sections from previous datasets. The idea is to test all new proposals against the default test corpus to check the extent to which they undo previous results. The test corpus is currently located in directory /language data working directory, which therefore also contains the output files associated with this corpus. The program version in the master branch will typically run this corpus.

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

Some sensory inputs map into meanings that are hard or impossible to construct (e.g., *democracy sleeps furiously*), while others are judged as outright ungrammatical. A realistic theory of language comprehension must appreciate these properties. The parser, when we abstract away from semantic interpretation, therefore defines a characteristic function from sensory objects into a binary (in some cases graded) classification of its input in terms of grammaticality or some related notion, such as semanticity, marginality or acceptability. These categorizations are studied by eliciting responses from native speakers. Any language comprehension model that captures this mapping correctly is said to be *observationally adequate*, to follow the terminology from (Chomsky 1964, 1965).

Some aspects of the comprehension model are language-specific, others are universal and depend on the biological structure of the human brain. A universal property can be elicited from speakers of any language. For example, it is a universal property that an interrogative clause cannot be formed by moving an interrogative pronoun out of a relative clause (as in, e.g., **who did John met the person that Mary admires_?*). On the other hand, it is a property of Finnish and not English that singular marked numerals other than 'one' assign the partitive case to the noun they select (*kolme sukkaa* 'three.sg.0 sock.sg.par'). Latter are acquired from the input during language acquisition. Universal properties plus the storage systems constitute the fixed portion of the parser, whereas language-specific contents stored into the memory systems during language acquisition and other relevant experiences constitute the language-specific, variable part. A theory of language comprehension that captures the fixed and variable components in a correct or at least

realistic way without contradicting anything known about the process of language acquisition is said to reach *explanatory adequacy*.

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

Suppose we have constructed a theory of language comprehension that is, or can be argued to be, observationally adequate and explanatory. Does it also agree with data obtained from neuro- and psycholinguistic experimentation? Realistic language comprehension involves several features that an observationally adequate explanatory theory need not capture. One such property concerns automatization. Systematic use of language in everyday communication allows the human brain to automatize recognition and processing of linguistic stimuli in a way that an observationally adequate explanatory model might or might not want to be concerned with. Furthermore, real language processing is sensitive to top-down and contextual effects that can be ignored when constructing a theory of language comprehension. That being said, the amount of computational resources consumed by the model should be related in some meaningful way with reality. If, for example, the parser engages in astronomical garden pathing when no native speaker exhibits such inefficiencies, then the model can be said to be insufficient in its ability to mimic real language comprehension. We say that if the model's computational efficiency and performance behavior is in general in line with that of real speakers, it is also *psycholinguistically adequate*. I will adopt this criterion in this study as well. Performance properties are discussed later in this document after we have examined the basic assumptions concerning grammar and grammatical representations.

Language production utilizes motoric programs that allow the speaker to orchestrate complex motoric sequences for the purposes of generating concrete speech or other forms of linguistic

behavior; language comprehension involves perception and not necessarily concrete motoric sequencing. Although there is evidence that perception involves (or “activates”) the motoric circuits and vice versa, overt repetition is (thankfully) not a requirement. A more interesting claim would be that the two systems share computational resources. This is the position taken in this study. Specifically, I will hypothesize, following (Phillips 1996, 2003) but interpreting that work in a slightly weaker form, that many of the core computational operations involved in language production and/or in the determination of linguistic competence are also involved in language comprehension. The same could be true of lower-level language processing, so that the motoric generation of, say, phonemes is linked in the human brain with systems that are responsible for the perception of the same units. What features are shared between the two modes of the language faculty, production and comprehension, is an empirical question we will settle by constructing detailed computational models of both and then by investigate what type of converge is possible.

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

The output contains a set of semantic interpretations. The output must constitute a set because the input can be ambiguous. The sentence *John saw the girl with a telescope* may mean that John saw a girl who was holding a telescope, or that John saw, by using the telescope that he himself had, the girl. This means that the original sensory input must be mapped to at least two semantic interpretations. These semantic interpretations must, furthermore, provide interpretations that agree with how native speakers interpret the input sentences.

The ultimate nature of semantic representations is a controversial issue. It becomes even more challenging when working within a fully computational theory. Whatever assumptions one makes must be captured in computational form. One way around this problem, adopted here, is to focus on selected aspects of semantic 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. We could require that the model provides that *John admires Mary* is processed so that John is judged the agent, Mary the patient, and not the other way around. The advantage is that we can predict semantic intuitions in a selective way without taking a strong stance towards the ultimate nature and implementation of the semantic notions. Another advantage is that we can focus on selected semantic properties without trying to understand how the semantic system works as a whole.

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

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

It is not possible to construct a model of language comprehension without assuming that there occurs a point at which something that is processed from the sensory input is used to construct a semantic interpretation. A model that does not posit any such a point would be incomplete, processing sensory inputs without generating any meaning. A syntax-semantic interface seems

to be a priori necessary on such grounds. Different theories make different claims regarding where they position that interface and what properties it has. It is possible to assume that the interface is located relatively close to the surface and operates with linguistic representations that have been generated from the sensory objects themselves by applying only few operations. Meaning would then be read off from relatively shallow linguistic representations. An alternative, a “deep” theory of the syntax-semantic interface, is a theory in which the sensory input is subjected to considerable amount of processing before anything reaches this stage. We can build the model by assuming that there is only one syntax-semantic interface, or several, or that the linguistic information flows into that system all at once, or in several independent or semi-independent packages. The only way to compare these alternatives, and many others that are imaginable, is to examine to what extent they will generate correct semantic interpretations for a set of input sentences; it is pointless to try to use any other justification or argument either in favor or against any specific model or assumption. In general, then, it makes no sense to try to posit any further restrictions or properties to the syntax-semantic interface apart from its existence.

3.2 Computational linguistics methodology

Scientific hypotheses must ultimately be justified by deducing the observed facts from the proposed hypothesis. The method is routinely used in all advanced sciences. We begin by narrowing down a dataset based on the interests of the particular study. In linguistics, a typical dataset contains grammatical and ungrammatical expressions paired with their meanings and other attributes, but it could involve actual use patterns, communicative intuitions, or pragmatic presuppositions. This dataset is captured by developing a hypothesis. Once the hypothesis is set up, an attempt will be made to calculate its empirical consequences that are compared with the dataset. The present theory is a formal theory in this sense. It consists of a set of assumptions or axioms, expressed and formalized in a machine-readable language, and the logical consequences of the assumptions or axioms are compared against empirical reality by letting the computer to perform the required calculations. The theory predicts grammaticality judgments, semantic interpretations and performance properties for any given set of natural language sentences, in any language.

The aim, then, is to provide a mathematical formula that is both sufficient and necessary to deduce data. Sufficiency is demonstrated by constructing the dataset from the hypothesis, as elucidated above. Necessity is more difficult to show, because it involves an additional concern of showing

that the proposed formula is also the simplest (in relation to the largest possible dataset).⁴ Although it is hard or impossible to show by relying on completely objective criteria that the hypothesis A is the “simplest” formula possible, if the notion can be made precise at all, we can compare two observationally adequate hypotheses A and B in terms of their simplicity in relation to some dataset. For example, if hypothesis A captures the dataset D by using an explicit table-lookup model where each input is paired by brute force with the correct output while hypothesis B relies on a general mechanism or rule, B will be voted as the favorite. This is because it generalizes to a much larger dataset; in fact, most linguistic hypotheses generalize over an infinite set of expressions and therefore supersede any finite model.

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

3.3 Overview of the hypothesis

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

⁴ These concerns have played a major role in recent minimalist grammars (Chomsky 1995, 2008). The research literature generated within this framework demonstrates how difficult it is to come up with an agreement on what counts as “simple” or “simplest.”

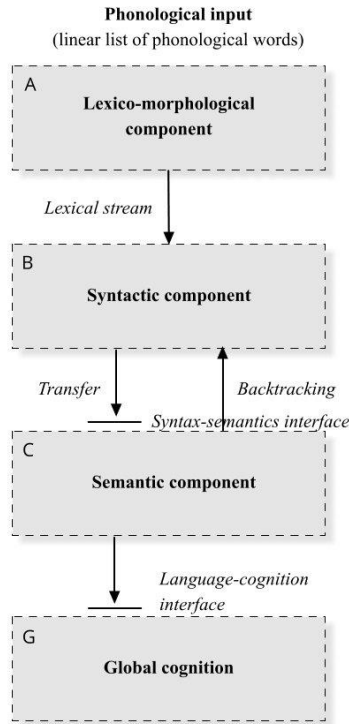


Figure 1. Main components of the model.

The input consists of a linear string of phonological words first processed by a *lexico-morphological component* (A) which retrieves corresponding *lexical items* from the lexicon and forwards them to the syntactic component (B) via a *lexical stream*. If the input word is ambiguous, the lexical items are put into a ranked list and explored in that order. Lexical items are sets of *lexical features*, which constitute the cognitive primitives of the model. The syntactic component (B) attaches incoming lexical items incrementally into a partial phrase structure representation in the current syntactic memory that has been assembled on the basis of the words seen so far. This process is illustrated in Figure 2. A left-to-right derivation of this type was first proposed by (Phillips 1996).

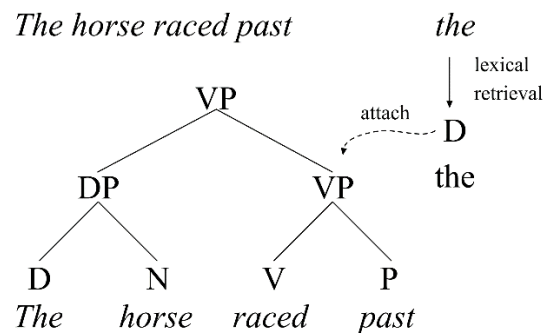


Figure 2. Operation of the syntactic component

The resulting phrase structure representation is called *spellout structure* since it corresponds transparently to the linear order in the sensory input, being directly generated from it.⁵ Once all words have been consumed from the input, the result will be *transferred* to the *syntax-semantic interface* (C) where the candidate representation is evaluated. If the syntactic interpretation is grammatical and interpretable, the solution will be *accepted* and the input string will be judged grammatical. An accepted structure is forwarded to a semantic component (called narrow semantics) which provides it with detailed semantic interpretation and interacts with global cognitive processes (thinking, decision making, discourse). This corresponds to a process in which the hearer “understands” the input sentence within some communicative context. If the candidate solution is not grammatical and/or cannot be interpreted semantically, it is *rejected* and no semantic interpretation results. In this scenario, the hearer has encountered a difficulty in understanding what the input sentence means. The syntactic component will be notified of the outcome, after which it begins to search alternative solutions by *backtracking*. This operation corresponds to a *reanalysis* of the input, in which the hearer will try to organize the words differently. As stated above, if no acceptable solution emerges, the hearer judges the sentence ungrammatical.

Suppose the input string is *the horse raced past the barn* and the syntactic module provides it with the syntactic representation [VP[DP *the horse*] [VP *raced* [PP *past* [DP *the barn*]]]. This input will be accepted at the syntax-semantics interface and interpreted as a declarative clause denoting a specific event containing the horse and the barn. If the input string contains an extra word *fell*, however, the first pass parse cannot be interpreted because there is no legitimate position for the last word *fell* (1).

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

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

(2) [VP [DP *the horse* [VP (that) *raced past the barn*]] *fell*]

⁵ Currently the linearly ordered input string and the spellout structure is mediated by a depth-first left-right linearization algorithm and its (ambiguous) inverse operation.

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

When the syntactic component assembles a solution for the input, the solution will be *transferred* to the syntax-semantics interface for evaluation and interpretation (see the arrow “Transfer” in Figure 1). While linguistic expressions are language-specific, the system that interprets them is universal. The cognitive capacities of speakers are virtually the same independent of the language(s) they happen to speak. Transfer removes language specific properties from the input so that it can be interpreted and processed further. It detects elements that occur in “wrong” positions where they cannot be interpreted and tries to *reconstruct* them into positions in which they can be interpreted. In Finnish, for example, speakers can reverse the order of the subject and object and produce an inverted OVS sentence that is noncanonical but still grammatical (Finnish is a canonical SVO language). Transfer will reconstruct the object and the subject into their canonical positions where they can be associated at the syntax-semantics interface with universal semantic notions such as agent and patient. In this case, the reconstruction is based on the overt morphological case features of the input words (nominative = subject, partitive = direct object). The process is illustrated in Figure 3, in a highly simplified form.⁶

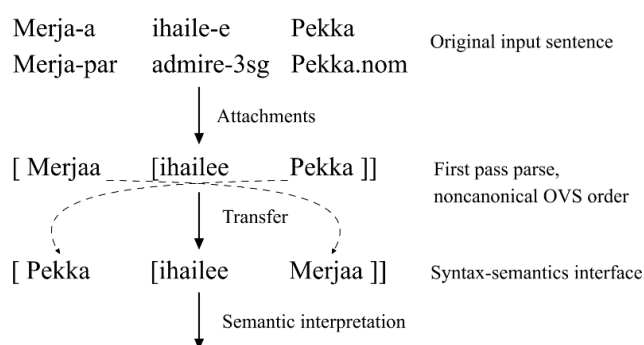


Figure 3. Transfer as an error correction mechanism.

Transfer is a cognitive reflex that is applied to all linguistic representations that are sent to the syntax-semantics interface for interpretation. We can imagine it as a noise tolerance mechanism

⁶ Transfer can be said to constitute a “reverse-engineered chain creation algorithm” within the context of modern generative grammar.

performing limited amount of reconstruction when the linguistic elements (words and their pars) appear at noncanonical positions.

Let us consider the nature of the semantic interpretation. At any given moment during a linguistic conversation or communication the hearer maintains a transitory repository of semantic objects called *global discourse inventory* that the conversation “is about.” Thus, if we talk about a person called John, the discourse inventory will contain a representation of John, the person. The objects maintained in the discourse inventory are language-external objects in the sense that they can be targeted by cognitive processes such as thinking even when no processing occurs in the syntactic pathway. When processing does occur in the syntactic pathway, it (like other sensory inputs) will cause changes in the discourse inventory. This corresponds to a situation in which the hearer updates his or her beliefs on the basis of the linguistic input. Semantic interpretation is ultimately viewed as a process in which the output of the syntactic pathway is converted into changes in the language-external discourse inventory. Each sentence adds, removes or updates elements and their properties in this repository. This conversion happens inside *narrow semantics*. The relationships between the syntactic pathway, narrow semantics and global cognition are illustrated in Figure 34.

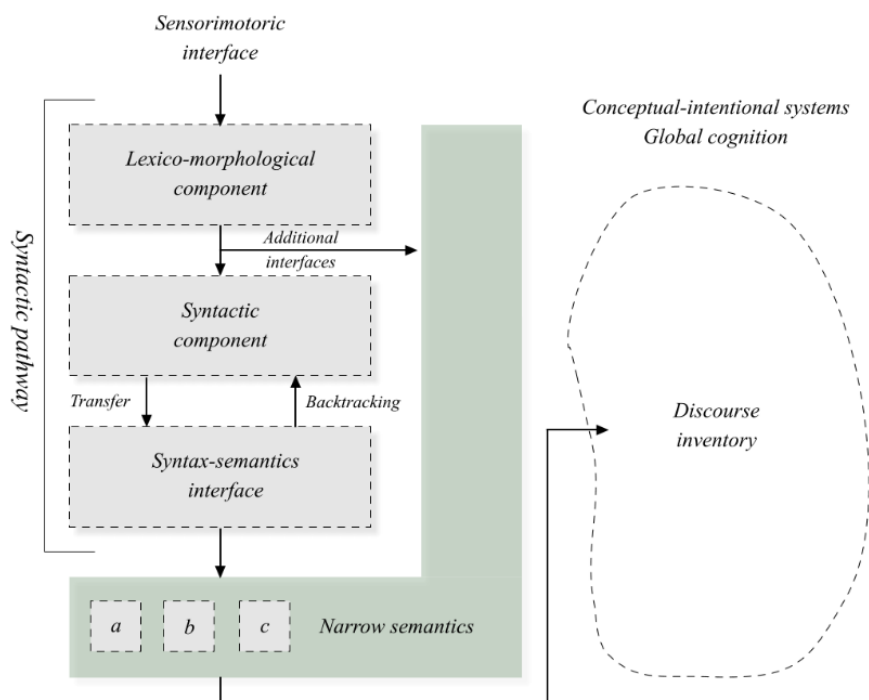


Figure 34. The syntactic pathway as embedded inside narrow semantics that mediates all communication between language and global cognition.

We can therefore conceptualize narrow semantics as a structure that mediates communication between the language faculty, the syntactic pathway more specifically, and the language-external systems that constitute global cognition and access the discourse inventory of transient semantic objects activated during the conversation or communication. A *conversation* is defined as a sequence of sentences that share the global discourse inventory, making it possible to use introduction sentences for populating the inventory and test sentences that make claims about those entities (e.g., *John₁ admires Mary₂; he₁ likes her₂*).

4 The kernel (comprehension cycle)

4.1 Introduction

This section provides the minimal specifications that makes it possible to build up the kernel of the model (the comprehension cycle) from scratch. The text is written for somebody working with a concrete algorithm or wants to understand the logic of the source code. Empirical matters dealing with the linguistic theory are discussed in published literature and are almost all omitted here. More detailed discussion that also involves the source code will be provided in Section 6. It is important to keep these two aspects, general ideas and implementation, apart, as the latter involves many details that are not relevant for the former. The same empirical hypothesis could be implemented in a variety of ways and by using any programming language. I will begin with the core recursive cycle implemented by the syntactic component (B, Figure 2) and then expand the discussion to other components.

4.2 Merge-1

Linguistic input is received by the hearer in the form of sensory stimulus. We can first think of the input, to simplify the situation first to the bare essentials, as a one-dimensional string $\alpha * \beta * \dots * \gamma$ 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 (Chomsky 1995, 2005, 2008), arranges them into a hierarchical representation. For example, if the input consists of two words $\alpha * \beta$, Merge yields $[\alpha, \beta](3)$.

(3) John * sleeps.

↓ ↓
[John, sleeps]

The first line represents the sensory input consisting of a linear string of phonological words, and the second line shows how these words are put together.

What do we mean by saying that they are “put together”? We assume that while the two words in the sensory input are represented as two independent objects, once they are “put together” in syntax in a manner shown in (3) they are represented as being part of the same linguistic chunk. Thus, at this point we can attend to both of them, manipulate them as part of the same representation, and in general perform operations that takes them both into account. Therefore, operation (3) presupposes that there exists a formally defined notion of phrase structure that is able to represent entities of this type.

Example (3) suggests that the syntactic component combines phonological words. While this is a possibility, it is not linguistically useful. We assume that (3) is mediated by *lexicon*: a storage of linguistic information that is activated on the basis of the original phonological words. The lexicon will map 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’). Thus, a word such as *John* is a set of features $\{f_1, \dots, f_n\}$, its phonological shape and meaning being among them. The lexicon and lexical retrieval are handled inside its own module. I will assume from this point on that syntax is operating with lexical items, not phonological words.

The resulting complex chunk $[\alpha, \beta]$ is asymmetric and has a *left constituent* and a *right constituent*. The terms “left” and “right” are mnemonic labels and do not refer to concrete leftness or rightness at the level of neuronal implementation. Their purpose is to distinguish the two constituents from each other. They are related to leftness indirectly: since we read the sensory input from left to right, (3) implies that the constituent that arrives first will be the left constituent. Thus, we can think of the left constituent as the “first constituent.” Because the configuration is asymmetric, it can be viewed as a list that can contain other lists as constituents.

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

(4) a. $[[\text{John } \textit{furiously}] \text{ sleeps}]$ b. $[[\text{John, sleeps}] \textit{furiously}]$ c. $[\text{John } [\text{sleeps } \textit{furiously}]]$

The operation illustrated in (4) differs in a number of ways from what constitutes the standard theory of Merge at the time of present writing. I will label the operation in (4) by Merge-1, symbol -1 referring to the fact that we look the operation from an inverse perspective: instead of generating linear sequences of words by applying Merge, we apply Merge-1 on the basis of a linear sequence of words in the sensory input and thus derive the structure backwards from left to right. The ultimate syntactic interpretation of the whole sentence is generated as a sequence of

partial phrase structure representations, first [*John*], then [*John, sleeps*], then [*John [sleeps, furiously]*], and so on, until all words have been consumed from the input.

Several factors regulate Merge-1. One concern is that the operation may in principle create a representation that is ungrammatical and/or uninterpretable. Alternative (4)(a) can be ruled out on such grounds: *John furiously* is not an interpretable fragment. Another problem of this alternative is that it is not clear how the adverbial, if it were originally merged inside subject, could have ended up as the last word in the linear input. The default left-to-right depth-first linearization algorithm would produce **John furiously sleeps* from (4)a. Therefore, this alternative can be rejected on the grounds that the result is ungrammatical and not consistent with the word order discovered from the input. If the default linearization algorithm proceeds recursively in a top-down left-right order, then each word must be merged to the *right edge* of the phrase structure, right edge referring to the top node and any of its right daughter node, granddaughter node, recursively. Under these assumptions a left-to-right model will translate into a grammatical model in which the grammatical structure is expanded at the right edge. A model of this type was first conceptualized by (Phillips 1996).

This leaves (4)(b) and (c). The parser will select one of them. Which one? There are many situations in which the correct choice is not known or can be known but is unknown at the point when the word is consumed. An incremental parsing process must nevertheless make decisions concerning an incoming word without knowing what the remaining words are. Let us assume that all legitimate merge sites are ordered and that these orderings generate a recursive search space that the algorithm will use to backtrack. The situation after consuming the word *furiously* will thus look as follows, assuming an arbitrary ranking:

(5) *Ranking*

- a. [~~[*John furiously*], *sleeps*]~~ (Eliminated)
- b. [[*John, sleeps*] *furiously*] (Priority high)
- c. [*John [sleeps furiously]*] (Priority low)

The parser will merge *furiously* into a right-adverbial position (b) and branch off recursively. If this solution does not produce a legitimate output, it will return to the same point and try solution (c). Every decision made during the parsing process is treated in the same way. All solutions constitute potential phrase structures, which are ordered in terms of their ranking.

The basic mechanism can be illustrated with the help of a standard garden path sentence such as (6).

- The horse raced past the barn.
- The horse raced past the barn fell.

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

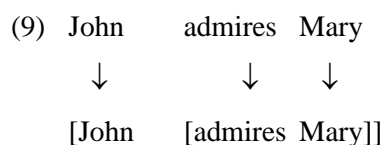
During the first stage, words *John* and *sleeps* are sisters. If the adverb is merged as the sister of *sleeps*, this no longer holds: *John* is now paired with a complex constituent [*sleeps furiously*]. If a new word is merged with [*sleeps furiously*], the structural relationship between *John* and this constituent is diluted further, as shown in (8).

One consequence of this is that if we merge two words as sisters, we cannot know if they will maintain the same or any close structural relationship in the derivation's future. In (8), they don't: future merge operations break up constituency relations established earlier. Consider the stage at which *John* is merged with a wrong verb form *sleep*. The result is a locally ungrammatical string **John sleep*. But because constituency relations can change in the derivation's future, we cannot rule this step locally as ungrammatical. It is possible that the verb 'sleep' ends up in a structural position in which it bears no meaningful linguistic relation with *John*, let alone one which would require them to agree with each other. Only those configurations or phrase structure fragments

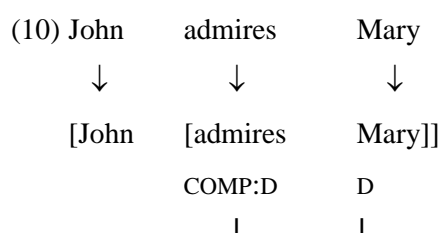
can be checked for ungrammaticality that cannot be tampered in the derivation's future. Such fragments are called *phases* in the current linguistic theory (Chomsky 2000, 2001). It is important to keep in mind when designing the algorithm that constituency relations established at point t do not necessarily hold at a later point $t + n$.

4.3 Lexical selection features

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



There is evidence that this derivation matches with the correct hierarchical relations between the three words. The verb and the direct object form a constituent that is merged with the subject. If we change the positions of the arguments, the interpretation is the opposite: Mary will be the one who admires John. But the fact that the verb *admire*, unlike *sleep*, can take a DP argument as its complement must be determined somewhere. Let us assume that such facts are part of the lexical items: *admire*, but not *sleep*, has a lexical feature !COMP:D which says that it requires a DP-complement. The fact that *admire* has the lexical feature !COMP:D can be used by Merge-1 to create a ranking based on an expectation: when *Mary* is consumed, the operation checks if any given merge site allows/expects the operation. In the example (10), the test passes: the label of the selecting item matches with the label of the new word.



Feature COMP:L means that the lexical item *licenses* a complement with label L, and !COMP:L says that it *requires* a complement of the type L. Correspondingly, –COMP:L says that the lexical item does *not* allow for a complement with label L. What the lexical features are is to some degree arbitrary. When the parser is trying to sort out an input, it uses lexical features (among other factors) to rank solutions. Remember, however, that these features cannot always be used to filter out solutions, because constituency relations can change in the derivation's future. But when the phrase structure has been completed, and there is no longer any input to be consumed, the same features can be used for filtering purposes. Filtering is performed at the LF interface (discussed

later) and will be called the *LF legibility test*. It checks if the solution provided by the parser makes sense from the semantic point of view.

Let us return shortly to the example with *furiously*. What might be the lexical features that are associated with this item? There are three options in (10): (i) complement of *Mary*, (ii) right constituent of *admires Mary*, and (iii) the right constituent of the whole clause. We can rule out the first by assuming (again, for the sake of example) that a proper name cannot take an adverbial complement. We are left with two options (11)a-b.

(11)

- a. [[_S John [admires Mary]] furiously]
- b. [John [_{VP} admires Mary] furiously]]

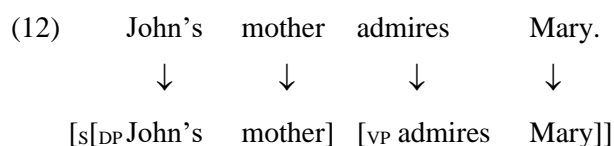
Independently of which solution is more plausible (or if they both are equally plausible), we can guide Merge-1 by providing the adverbial with a lexical selection feature which determines what type of left sisters it is allowed or is required to have. I call such features *specifier selection features*. A feature SPEC:S (“select the whole clause as a specifier/sister”) favors solution (a), SPEC:V favors solution (b). What constitutes a specifier selection feature will guide the selection of a possible left sister during comprehension. Because constituency relations may change later, we do not know which elements will constitute the *actual* specifier-head relations in the final output. Therefore, we use specifier selection to guide the parser in selecting left sisters, and later use them at the syntax-semantics interface (LF interface) to verify that the output contains proper specifier-head relations.

Some languages such as Finnish and Icelandic require that the specifier position of the finite tense is filled in by some phrase, but it does not matter what the label of that phrase is. Instead of providing a long list of specifier features, we capture this situation by an unselective specifier feature -SPEC:*, SPEC:* and !SPEC:*. Because the feature is unselective, it is not interpreted thematically, it cannot designate a canonical position, and hence the existence of this feature on a head triggers \bar{A}/A movement reconstruction (Section 4.8.1). This constitutes a sufficient (but not necessary) feature for reconstruction. Corresponding to SPEC:* we also have !COMP:*, which is a property all functional heads have, possibly by definition.

In sum, we use lexical features for guiding the parsing process towards meaningful solutions and later for checking that any given solution is grammatical and/or can be interpreted. Lexical features are provided, of course, in the lexicon.

4.4 Phases and left branches

Let us consider the derivation of (12).



After the finite verb has been merged with the DP *John's mother*, no future operation can affect the internal structure of that DP. This is because Merge-1 is always to the right edge. All left branches therefore become *phases* in the sense of (Chomsky 2000, 2001), in that the derivation can effectively “forget” them inside that particular parsing path. This left branch phase condition, as we called it, was argued for by (Brattico and Chesi 2020) and then adopted into all subsequent models. We can formulate the condition tentatively as (13), but a more rigorous formulation will be given as we proceed.

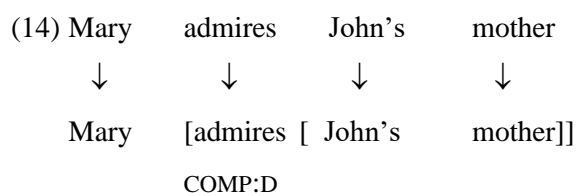
(13) *Left branch phase condition*

Derive each left branch independently.

All left branches are effectively thrown away from the cognitive working space once they have been assembled and fully processed. If no future operation is able to affect a left branch inside the that parsing path, all grammatical operations (e.g., movement reconstruction) that must be done in order to derive a complete phrase structure must be done to each left branch before they are sealed off. Furthermore, if after all operations have been done the left branch fragment still remains ungrammatical or uninterpretable, the original merge operation that created the left branch phase must be cancelled. This limits the set of possible merge sites. Any merge site that leads into an “unrepairable” left branch can be either filtered out as unusable or at the very least be ranked lower. Keep in mind, however, that since the model is able to backtrack, it is able to shift into another parsing path and thus reconsider the left branches in that new path.

4.5 Labeling

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



The verb's complement selection feature refers to the label D of the complement. What is the relationship between the label D and the phrase *John's mother* that occurs in the complement position of the verb in the above example? That relationship is defined by (15).

(15) *Labeling*

Suppose α is a complex phrase. Then

- a. if the left constituent is primitive, it will be the label; otherwise,
- b. if the right constituent is primitive, it will be the label; otherwise,
- c. if the right constituent is not an adjunct, apply (15) recursively to it; otherwise,
- d. apply (15) to the left constituent (whether adjunct or not).

The algorithm searches for the closest possible primitive head from the phrase starting from the top, which will then constitute the label. Here "closest" means closest from the point of view of the selector; if we analyze at the situation from the point of view of the labeled phrase itself, then closest is the highest or most dominant head. Conditions (15)c-d mean that labeling ignores right adjuncts (this will be a defining feature of adjuncts). Example (16) shows some basic examples.

- (16) a. [VP admires Mary] = left primitive verb;
 b. [PP from Mary] = left primitive preposition;
 c. [VP [DP That man][VP admires Mary]] = verb is the first left primitive;
 d. [DP the man] = definite determiner is left, hence the label;
 e. [VP[VP admires Mary]<furiously>] = adjunct is ignored;
 f. [NP [DP that man's] car] = right primitive noun.

Since the phrase structure geometry can change during incremental parsing, also labeling may change. This is shown by the following hypothetical parsing derivation:

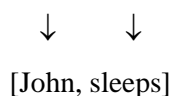
- (17) a. [DP the man] + admires =
 b. [VP [DP the man] V].

What was originally a DP was transformed into a VP when the verb was merged. It follows that we do not need to project or fit phrase structure templates to the input; rather, we let the input to generate an appropriate phrase structure. Finally, it is important to keep in mind that labels are not intrinsic features of the phrase structure. They are determined dynamically during the derivation.

The term *complex constituent* is defined as a constituent that has both the left and right constituent; if a constituent is not complex, it will be called *primitive constituent*. A constituent that has only

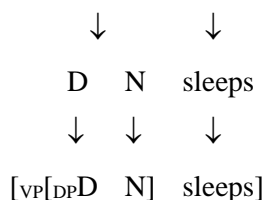
the left or right constituent, but not both, will be primitive according to this definition. Consider again the derivation of (3), repeated here as (18).

(18) John + sleeps.



If *John* is a primitive constituent having no left or right daughters, labeling will categorize [*John sleeps*] as a DP or NP. The left constituent will be the label, thus NP = [N sleeps]. This is wrong: (18) is a sentence or verb phrase, not a DP. We therefore assume that *John* is a complex constituent despite of its appearance. Its structure is [_{DP} D N]. This information comes from the lexicon, where proper names are decomposed into D + N structures. The structure of (18) is therefore (19).

(19) John + sleeps.



4.6 Upward paths and memory scanning

The words and constituents making up phrase structure representations can enter into several types of mutual dependency relationships. To illustrate, consider the following pair of Finnish sentences.

- (20) a. Pekka **ei** ihailut *Merja-n/ Merja-a.
 Pekka not admire Merja-ACC Merja-PAR
 'Pekka did not admire Merja.'
- b. Pekka **on** ihailut Merja-n/ *Merja-a.
 Pekka has admired Merja-ACC Merja-PAR
 'Pekka did admire Merja.'

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

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

This example shows that the relationship between the polarity element *not/is* and the case form cannot depend on local selection; rather, it spans “through” the phrase structure. Indeed, many dependencies of this type are nonlocal, which means that they can cover an unbounded distance in any given structure. They are modelled by relying on the notion of *path*. It is assumed, first, that the case form at the case assignee – word *Merja-a* ‘Merja-PAR’ – must enter into a compatibility check with another element, the case assigner. In this case the case forms accusative and partitive enter into a compatibility check with the polarity element. It does this by forming an *upward path* from the case assignee to the case assigner, in effect by searching the latter. We can define the notion as follows. Suppose α is an element requiring checking; then

(22) *Upward path*

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

For example, if the phrase structure is (23)

- (23) [_{AP} ... A [_{BP} ... B [_{α P} ... α ...]]]

then the upward path from α contains α P, BP and AP (the dominating constituents) plus their immediate constituents A and B. The dependency itself is formed by literally scanning through the path for a *target element*, and the first target element available is always selected. If the target is not found, the checking operation fails. The intuitive idea behind the scanning operation is that when α triggers the operation, then the elements defined by the path (23) constitute the portion of the grammatical structure that is inside the active syntactic working memory at that moment. In other words, the system is scanning the contents of the active syntactic working memory when some elements triggers checking. Case forms constitute one subclass of things in language that trigger the checking (scanning) operation; there are many others.

4.7 Adjunct attachment

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

(24) (Finnish)

- a. *Ilmeisesti* Pekka ihailee Merjaa.

- apparently Pekka admires Merja
 'Probably Pekka admires Merja.'
- b. Pekka *ilmeisesti* ihailee Merjaa.
 Pekka apparently admires Merja
- c. Pekka ihailee *ilmeisesti* Merjaa.
 Pekka admires apparently Merja
- d. ??Pekka ihailee Merjaa *ilmeisesti*
 Pekka admires Merja apparently

The adverbial *ilmeisesti* 'apparently' can occur almost in any position in the clause. Consequently, it will be merged into different positions in each sentence. This confuses labeling and present a challenge for the parser, because the position of the adverb is unpredictable. The problem is to define what this type of 'free attachment' means.

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

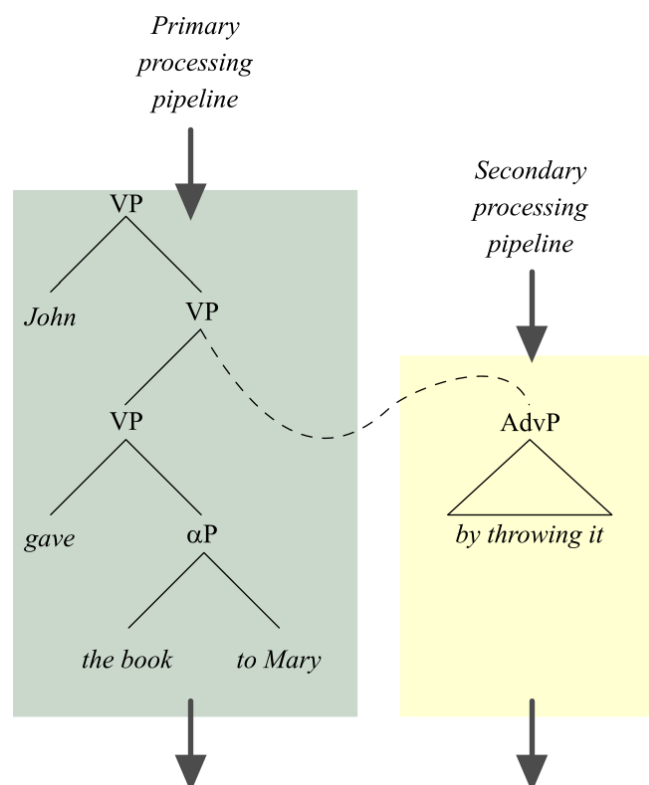


Figure 24. A nontechnical illustration of the way the linear phase comprehension algorithm processes adjuncts. Adjuncts are geometrical constituents that are "pulled out" of the primary working memory and are processed inside separate processing pipeline. We can imagine that the VP is made up of two "segments."

Thus, the labeling algorithm as specified in Section 4.5 ignores adjuncts. The label of (25) becomes V and not Adv: while analyzing the higher VP shell, the search algorithm does not enter inside the adverb phrase; the lower VP is penetrated instead.

(25) [_{VP} John [_{VP} [_{VP} admires Mary] <_{AdvP} furiously>]]

Consider (26) next.

(26) 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 the notion of sisterhood to ignore 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. From the point of view of labeling, selection and sisterhood, then, the structure of (26) is [_{VP} [_{DP} John] sleeps]. The fact that adjuncts, like the adverbial *furiously* here, are optional now follows from the fact that they are automatically excluded from selection and labelling. Whether they are present or absent has no consequences for either of these dependencies.

It follows from these assumptions, however, that adjuncts could be merged anywhere, which is not correct. It is assumed that each adverbial (head) is associated with a feature linking it with a feature or set of features in its hosting, primary structure. In this case the linking relation is established by a *tail-head relation*. For example, a VP-adverbial is linked with V, a TP-adverbial is linked with T, a CP-adverbial is linked with C. Linking is established by checking that the adverbial (i) occurs inside the corresponding projection (e.g., VP, TP, CP) or is (ii) can be linked with the corresponding head by means of an upward path. Conditions (i-ii) have slightly different content and are applied in different circumstances.

(27) *Condition on tail-head dependencies*

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

- a. α occurs inside a projection whose head β_F , or HP is β_F 's sister;
- b. α can be linked with β_F through an upward path (22).

β_F denotes a head (or phrase) that has feature F. Condition (27)(a) is relatively uncontroversial. It states that a VP-ad adjunct must occur inside VP, or more generally, α P adjunct has to be inside a projection from α , where α is some feature. It does not restrict the position at which an adjunct must occur inside α P. Therefore, both right-adjoined and left-adjoined phrases are accepted. Condition (27)(b) allows some adjuncts, such as preposition phrases, remain in a low right-adjoined or in “extraposed” positions in a canonical structure. If an adverbial/head does not satisfy

a tail feature, it will be reconstructed into a position in which it does during transfer. This operation will be discussed in Section 4.8.6.

4.8 Transfer

4.8.1 Introduction

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

4.8.2 Phrasal reconstruction: Some general comments

A phrase or word can occur in a *canonical* or *noncanonical* position. A canonical position *in the input string* could be defined as a position such that, given just the regular merge-1 operations, the constituent ends up in a syntactic position where it passes all LF interface tests. For example, regular referential or quantificational arguments must occur inside the verb phrase at the LF interface in order to receive thematic roles and satisfy selection properties of the verb. Example (28) illustrates a sentence where this is the case.

(28) John	admires	Mary
↓	↓	↓
[_{VP} John	[_{VP} admires	Mary]]

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

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

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

(30) [_{VP} who [_{VP} Pekka admires]]

This solution violates two selection rules: there are two specifiers at the left edge, namely *who* and *Pekka*, and *admire* lacks a complement. Furthermore, even if the parser backtracks, it cannot find a legitimate solution. A candidate structure [_{VP} [_{DP} *who Pekka*] *admires*] is also illegitimate.⁷ The two violations are related to each other, however: the element that triggers the double specifier violation at the left edge is the same element that should be at the complement position of the verb. The interrogative pronoun causes these problems because it has been “dislocated” (by the speaker) to the noncanonical position from its canonical complement position.

Dislocation is a systematic feature of all or most natural languages. In Finnish, for example, almost all word order variations of the referential arguments of a finite clause are possible. 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 orders only create “stylistic” differences between the sentences. For example, the two orders in (31) correspond to the same core proposition ‘Pekka admires Merja’.

- (31) a. Pekka ihailee Merja-a. (SVO)
 Pekka.NOM admires Merja-PAR
 ‘Pekka admires Merja.’
 b. Merja-a ihailee Pekka. (OVS)
 Merja-PAR admires Pekka.NOM
 ‘Pekka admires Merja.’

It is obvious, then, that something must happen to these sentences before they are interpreted semantically. Unless the system reconstructs some type of normalized form where all selection restrictions can be checked, we cannot filter out ungrammatical sentences from the grammatical ones. It is clear, furthermore, that Finnish speakers use overt case forms, here the nominative and partitive, to do this. They understand intuitively that the partitive argument must represent the object even if it occurs in the preverbal subject position, and that the postverbal nominative argument represents the agent. In short, then, the syntactic parser must reverse-engineer the

⁷ We could feed (30) directly to the LF interface component. This representation does not satisfy the complement selection features of the verb *admire*. If we do not control for what is filling the complement position of the verb, the model will not be able to separate grammatical sentences correctly from the ungrammatical ones. It would accept **John admires* or even **John admires Mary to leave*. There must therefore occur a process which relates the first element of the clause with the empty position at the end of the clause. Similarly, the main verb is preceded by two argument DPs, yet we cannot allow this to happen generally. A sentence such as **John Mary admires* is ungrammatical.

construction in order to create a representation that can be interpreted at the LF interface. These operations take place during transfer.

Almost all reconstruction operations are based on the same computational template. First the system recognizes that the element occurs in an illegitimate or deviant position where it cannot satisfy one or several LF interface condition(s), such as selection or say case checking. Once the system detects the presence of an offending constituent, it will attempt reconstruction, where a legitimate position is sought. If a legitimate position is found, the element is copied there, and the search ends; if a legitimate position is not found, no operation applies and the constituent remains in its surface position. In that case, some later operation may still dislocate it. How much variation is allowed in any given language depends on the properties of reconstruction and transfer: it is possible that transfer fails to recover the expression, resulting in a strong feeling of deviance or ungrammaticality.

If the LF-interface properties are virtually universal and do not vary between languages, then we must assume that the surface properties reconstructed by transfer vary, and that it is the source of all language-specific properties. We will indeed think of the surface variations as different ways linguistic information can be “packaged” for communication through the sensorimotoric interfaces. It constitutes language-specific noise.

4.8.3 *Ā-chains*

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

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

To illustrate how transfer and reconstruction works, I will again make some simplifications. Let us assume that merge-1 creates the structure shown on the third line, and that the finite verb is mapped to a single verb V. As we already noted, there are two problems with this result: the verb has two specifiers, two ‘agents’ if you will, and no complement or patient. One of these two conditions can trigger reconstruction. In the current algorithm, it is triggered by the presence of an extra specifier *ketä* ‘who.par’. The reconstruction algorithm then searches for the first position for this element where it can generate a legitimate position, and discovers solution (33).

- (33) [~~who~~ [Pekka [admires who]]]

This representation is shipped to the LF-interface. The interrogative pronoun is copied from SpecVP into CompVP, and will now be interpreted as the patient of the verb *admire*. The upper

copy is ignored for most subsequent computations. The search algorithm proceeds from the deviant element downstream until either a suitable position is found or there is no more structure. It cannot go upwards or sideways. In this case the operation begins from the sister of the targeted element and proceeds downward while ignoring left branches (phases) and all right adjuncts. This operation is called *minimal search*, following (Chomsky 2008). It is discussed in more detail in Section 4.8.7. The element is copied to the first position in which (i) it can be selected, (ii) is not occupied by another legit element, and in which (iii) it does not violate any other condition for LF objects. If no such position is found, the element remains in the original position and may be targeted by later operation. If a position is found, it will be copied there; the original element will be tagged so that it will not be targeted second time.

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

- (34) a. Who admires Pekka?
 [_{VP} who_{wh} [_{VP} admires_{wh} Pekka]]
 b. who Pekka admires?
 ~~who~~_{wh} C_{wh} [Pekka [admires who_{wh}]]

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

- (35) a. John asks/claims + [_{VP+wh} who [admires_{V+wh} Pekka]
 b. John asks/claims + [_{CP+wh} ~~who~~ C_{wh} [Pekka [admires who]]]

The verb *claim* cannot select a sentence headed by the *wh*-feature, while the *ask* must. In the case of (b) it is the reconstruction algorithm that is responsible for the generation of an extra C head into the structure. We can think of this operation as reconstructing a phonologically null head, i.e. a head that had no direct representation in the sensorimotoric input.

The dislocated phrase also represents the *propositional scope* of the question. The interpretation is one in which the speaker is asking for the identity of the object of admiration (‘which x: Pekka admires x’). Let us consider how dislocation represents the propositional scope of an interrogative clause. First, the reconstructed interrogative pronoun or phrase is called an *operator*. This just 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 the hearer is assumed to fill in. Then, any finite element that contains the same operator feature, in this case *wh*-feature, will determine the scope.

(36) ~~who~~_{wh} C_{wh,fin} does John admire who_{wh}
└──────────────────┘

The operator-scope dependency, marked by the line in the representation above, is checked by an operator-variable module inside narrow semantics, which pairs the operator with its scope marker. If feature OP (e.g. *wh*) represents the operator, then the closest head with [FIN][OP] will be the scope marker. These dependencies, which have both a syntactic and semantic dimension, are called *\bar{A} -dependencies* or *\bar{A} -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; 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.⁸

Crosslinguistic studies shows that 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; it is just one of the many options.

⁸ 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) which understands what these features mean and how their mutual dependencies should be computed.

4.8.4 *A-chains and EPP*

In addition to \bar{A} -chains, discussed in the previous section, many languages exhibit another type of phrasal reconstruction, called A-chains. In English, for example, the preverbal subject position can be occupied by both the agent and patient.

- (37) a. John admires Mary.
 [John [admires Mary]]
 b. Mary was admired (by John).
 [Mary [Aux [admired (by John)]]]

The direct merge-1 solution (second lines in the above example) is again wrong in the case of (b), because here Mary would be in the agent position, John as the patient (if present). However, we can conclude on the basis of data of this type (a-b) that the English preverbal subject position is *not* a thematic position after all. It can be occupied by both agents (a) and patients (b). This is supported further by the data from Finnish, which shows that argument order does not correlate necessarily with thematic interpretation. This, then, brings us to the presence of the tensed auxiliary verb *was* in the example (b). We have not yet analyzed elements of this type, but now the matter becomes crucial. Note that both the bare finite verb (a) and the aux-verb construction (b) involve a tensed finite verb; the difference is that when the sentence contains an auxiliary, tense information is expressed by the auxiliary. Let us assume, therefore, that tense information is an independent packet of information in the sentence that may be expressed either by combining it with the verb (a) or by carrying it by an otherwise dummy auxiliary verb (b). We can depict the situation by using the schema (38).

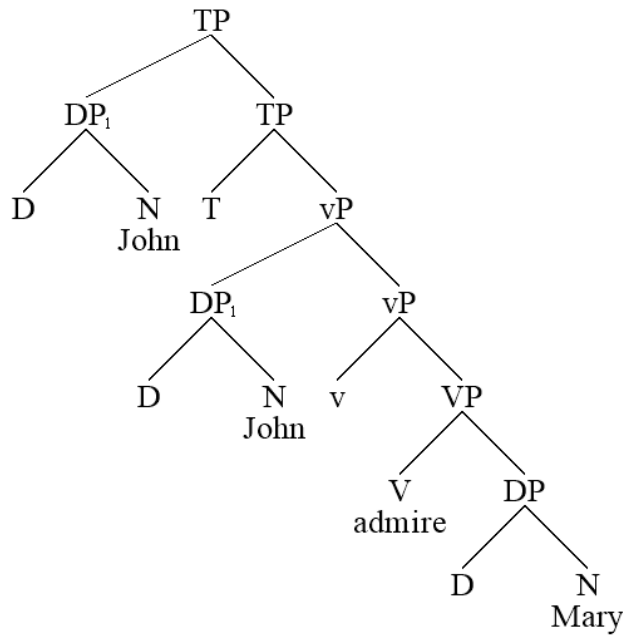
- (38) [TP John [TP T_{pst/prs} [VP admire Mary.]]]

The tense T can be expressed either by an auxiliary (*was*, *does*) or by combining it with the main verb; the latter operation will be discussed in a subsequent section. What matters here, however, is the fact that now 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: we assume that the preverbal subject position is SpecTP and the thematic agent position SpecVP, and assume that the grammatical subject is reconstructed from the former into the latter (39).

- (39) [TP ~~John~~ [TP T [VP John [VP admire Mary.]]]

This operation called *A-reconstruction* and it forms an *A-chain*. The operation is triggered by the fact that SpecTP is not a thematic position – hence the referential argument cannot remain at this position at the LF-interface – and the operation locates the first position where it can be interpreted (40).

(40)



The fact that SpecTP is a nonthematic position is marked in the current implementation by the so-called EPP feature that is inside (English) T, so it is this feature, then, that activates A-reconstruction.⁹ A-reconstruction differs from \bar{A} -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.

4.8.5 Head reconstruction

Head reconstruction is involved in the handling of morphologically complex heads. Consider against the finite verb construction discussed in the previous section, repeated in (41).

(41) John admires Mary.

We 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

⁹ The standard EPP feature has an additional effect: it forces an element into the specifier position.

Therefore, it is implemented by a specifier selection feature [*!SPEC:**], but this is not relevant here and controversial in any case.

by the finite verb, but what was left unexplained was how the finite verb is dissolved into the two independent components when that strategy is used. Head reconstruction solves this issue.

First we should note that whether and how grammatical heads are chunked into morphological and phonological words is to some extent arbitrary and subject to crosslinguistic variation. Thus in English, we can express the same tense information by *John does admire Mary* and *John admires Mary*, where in the latter the two heads have been chunked together. Head reconstruction must therefore contain an operation that recovers morphological chunks from phonological words. This part is handled by the lexico-morphological module. It takes phonological words as input and decomposes them into their constituent morphemes, thus /admires/ $\sim T_{\text{prs}} + V$. Second, these heads must be “streamed” into the syntactic component so that they can be merged-1 to the syntactic structure that is being assembled. The end result of this operation is that a *complex head* $T(V)$ is created inside the syntactic component. Thus, the lexical streaming operation maps lexical decompositions into complex heads: $T + V \sim T(V)$. Finally, syntactic head reconstruction extracts the heads and distributes them into the structure, $T(V) \sim [T \dots [...V\dots]]$. The steps are illustrated in (42).

- (42) John admires Mary. (Input)
 [John [T(V) 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 (42) is selected because T can select a VP complement.

Complex heads, however, are grammatical objects that do not exist in the phrase structure system as defined so far. We just stipulated them into existence in the above example. Let us consider their properties in some more detail. We have assumed so far that the phrase structure is made up of *primitive constituents*, which are elements that have zero daughters, and complex constituents, which have two daughters. Thus, if α and β are primitive constituents, then $[\alpha \beta]$ is a legitimate complex constituent. Because $[\alpha \beta]$ will always be treated like a phrase, we can also say that it is a *phrasal constituent*. What we have not considered is a situation where the complex constituent has only one daughter constituent (left or right). This configuration creates complex constituents that are *nonphrasal*, which are the complex heads referred to above. Thus we have $\alpha(\beta) = [\alpha \beta]$, where α is a constituent that has only one (right) constituent β . The mechanism is iterative, and allows the system to chain several heads into ever more complicated words, for example $A(B(C))) = [A [B C]]$. 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.

These technical assumptions are based on a simple intuition. The intuition is that morphologically complex words are linear sequences of (sensorimotoric) elements, and that the notation $[\alpha \beta]$ represents one such sequence. We can think of the lopsided constituency relation as representing simple “followed by” relation: produce α followed by β , in short $\alpha * \beta$. Each element in the sequence must be atomic and map into one sensorimotoric program. Thus, an organism that only has access to this mechanism can produce sequences of atomic symbols (perhaps akin to a bird song). Phrasal constituents $[\alpha \beta]$ are exactly like this – produce α followed by β – with the exception that we have recursion and not linear only lists: α and β can be either linear sequences or complex phrasal constituents. Expressed in terms of sensorimotoric processing, it expresses the intuition that sensorimotoric programs can be based on hierarchical plans. Head reconstruction and its mirror, head movement, map between the two formats as required by the lexicon of the language. At the universal LF-interface level, all independent grammatical packages must be represented as such; at the sensorimotoric interface some of these packages may or must be expressed in a condensed, “chunked” or (evolutionarily) more primitive form.

4.8.6 *Adjunct reconstruction*

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

(43)

- a. Pekka käski meidän ihailla Merjaa.
 Pekka.nom asked we.gen to.admire Merja.par
 [Pekka [asked [we [to.admire Merja]]]]
 ‘Pekka asked us to admire Merja.’
- b. Merjaa käski meidän ihailla Pekka.
 Merja.par asked we.gen to.admire Pekka.nom
 [Merja [asked [we [to.admire Pekka]]]
 ‘Pekka asked us to admire Merja.’

Derivation (b) is incorrect. Native speakers interpreted the thematic roles as being identical in these examples. The subject and object are in wrong positions. Neither \bar{A} - nor A-reconstruction can handle these cases. The problem is created by the grammatical subject *Pekka* ‘Pekka.nom’, which has to move upwards/leftward in order to reach the canonical LF-position SpecVP. Because the distribution of thematic arguments is very similar to the distribution of adverbials in Finnish,

I have argued that richly case marked thematic arguments can be promoted into adjuncts.¹⁰ Case forms, according to this view, are viewed as morphological reflexes of tail-head features. If the condition is not checked by the position of an argument in the input, then the argument is treated as an adjunct and reconstruction into a position in which the condition is satisfied. In this way, the inversed subject and object can find their ways to the canonical LF-positions (44). Notice that because the grammatical subject *Pekka* is promoted into adjunct, it no longer constitutes the complement; the partitive-marked direct object does.

- (44) [\langle Merjaa \rangle_2 T/fin [$__1$ [käski [meidän [ihaila [$__2$ \langle Pekka \rangle_1]]]]]
 Merja.par asked we.gen to.admire Pekka
 'Pekka asked us to admire Merja.'

4.8.7 Minimal search

Reconstruction uses minimal search for locating reconstruction sites. The term minimal comes from the fact that the first acceptable position, whether it is ultimately right or wrong, is always selected. 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.¹¹ The operation therefore moves downwards on the right edge of the phrase structure and follows selection and labelling. Left and right (adjunct) phrases are ignored. The operation never branches since the algorithm provides a unique solution for any constituent. To take a simple example for illustration, consider (45).

- (45) Who does John's brother admire $__1$ every day?

To reach the reconstruction site $__1$, the search algorithm must avoid going into the subject and into the temporal adverbial. This is prevented by minimal search, which follows the projectional spine of the sentence and ignores both left specifiers/adjuncts and right adjuncts (there are no "right specifiers" in the model).

There is a possible deeper motivation for minimal search. Both left branches and right adjuncts are transferred independently as phases and therefore they are no longer in the current syntactic working memory. It seems, in other words, that minimal search coincides with the contents of the current syntactic working memory at any point in the derivation. If this hypothesis can be maintained, then we could replace the current definition with 'search the contents of the syntactic

¹⁰ See (Brattico 2016, 2018, 2020, 2022) and (Baker 1996; Chomsky 1995: 4.7.3; Jelinek 1984) for earlier formulations of the same idea.

¹¹ Notation $\{\alpha \beta\}$ refers to $[\alpha \beta]$ or $[\beta \alpha]$, depending on the actual case.

working memory in top-down order’. This hypothesis has not been explored or tested as of present writing, but the working memory mechanism as such already exists.

4.8.8 Agree-1

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

- (46) a. John admires Mary
 b. *John admire Mary

The third person features of the subject are reflected in the third person agreement marker *-s* on the finite verb. The term agreement or phi-agreement refers to a phenomenon where the gender, number or person features (collectively called phi-features in this document) of one element, typically a DP, covary with the same features on another element, typically a verb, as in (46).

Before moving forward, it is important to note that not all lexical elements express phi-features, and those which do can be separated into three classes with respect to the type of agreement that they exhibit. Whether a lexical item exhibits agreement is determined by lexical feature $\pm\text{VAL}$. A lexical item with $-\text{VAL}$ does not exhibit phi-agreement. In English, conjunctions (*but*, *and*) and the complementizer (*that*) belong to this class, and are marked for $-\text{VAL}$. Those lexical items which can exhibit agreement have feature $+\text{VAL}$. This feature therefore triggers Agree-1. Heads which phi-agree can be divided further into two groups: those which exhibit full phi-agreement with an argument and those which exhibit concord. Whether a lexical item exhibits full phi-agreement with a full argument is determined by feature $\pm\text{ARG}$. Negative marking $-\text{ARG}$ creates concord, the positive marking $+\text{ARG}$ forces the element to get linked with a full argument DP. This linking will be interpreted at LF interface as *predication*. These features leave room for a predicate that is linked with an argument but does not phi-agree. This group creates control, discussed in the next section. The four options are illustrated in Table 1.

Table 1.

Four agreement signatures depending on features $\pm\text{VAL}$ and $\pm\text{ARG}$.

	$-\text{VAL}$	$+\text{VAL}$
$-\text{ARG}$	Lexical items exhibiting neither agreement nor require arguments (particles, such as <i>but</i> , <i>also</i> , <i>that</i> , <i>not</i>)	Lexical items which exhibit agreement but do not require linking with arguments (agreement by concord, e.g. <i>piccolo</i> , <i>pienet</i>)
$+\text{ARG}$	Lexical items which do not exhibit agreement but require linking with arguments (control constructions, such as <i>to leave</i> , <i>by leaving</i>)	Lexical items which exhibit agreement and linking with arguments (finite verbs, <i>admires</i>)

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 extralinguistic world. Thus, *Mary* refers to a third person singular individual. A predicate, on the other hand, must be linked with an argument that has phi-features. To model this asymmetry, a predicate with +ARG will have *unvalued* phi-features, denoted by $\varphi_$. The value will be provided by the argument with which the predicate is linked with. This is shown in (47).

(47) Mary	[admires	John]
D N	T/fin	
↓	↓	
PHI:NUM:SG	PHI:NUM:_	
PHI:PER:3	PHI:PER:_	
PHI:DET:DEF	PHI:DET:_	
	+ARG, +VAL	

The operation that fills the unvalued slots for the predicate is *Agree-1*. It values the unvalued features of a predicate on the basis of an argument with which the predicate is linked with (48). Recall that *Agree-1* is applied only to heads with +VAL. This ensures that only selected words, such as finite verbs, exhibit agreement.

(48) Mary	[admires	John]
PHI:NUM:SG	PHI:NUM:SG	
PHI:PER:3	PHI:PER:3	
PHI:DET:DEF	PHI:DET:DET	
└─ Agree-1 ─┘		

As a consequence of *Agree-1*, the unvalued features disappear from the lexical item and the predicate is now linked with its argument. If no suitable argument is found, the unvalued features remain in place. This means that the LF interface will confront unvalued and uninterpretable phi-features. This scenario will be discussed in the next section.

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

(49) Adoro Luisa.
 admire.1sg Luisa
 ‘I admire Luisa.’

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

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

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

- (51) a. admire- = lexical predicate, hence it has unvalued phi-features $_ \varnothing$;
 b. -s = third person singular valued inflectional phi-features \varnothing .
 c. = $_ \varnothing + \varnothing$.

Unsaturated features will still require valuation, which triggers Agree-1. The existing valued phi-features, if any, impose two further consequences to the operation. First, Agree-1 must check that if the head has valued phi-features, no phi-feature conflict arises. Thus, a sentence such as **Mary admire John* will be recognized as ungrammatical by Agree-1. Second, we allow Agree-1 to examine the valued phi-features inside the head if (and only if) no overt phrasal argument is found. The latter mechanism will create subjectless pro-drop sentences. We thus interpret a valued phi-set inside a head as if it were a truncated pronominal element (52).

(52) adoro Luisa.
 admire.1sg Luisa
 admire.pro Luisa
 ‘I admire Luisa.’

Agree-1 is limited to local domain. The local domain is defined by (i) its sister and specifiers inside its sister; (ii) its own specifiers; and (iii) the possible truncated pro-element inside the head itself, in this order. The first suitable element that is found is selected.

4.8.9 Ordering of operations

\bar{A}/A -reconstruction and adjunct reconstruction presuppose head reconstruction, because heads and their lexical features guide \bar{A}/A - and adjunct reconstruction. The former relies on EPP features and empty positions, whereas the latter relies on the presence of functional heads. Furthermore, \bar{A}/A -reconstruction relies on adjunct reconstruction: empty positions cannot be recognized as such unless orphan constituents that might be hiding somewhere are first returned to their canonical positions. The whole sequence is (53).

(53) Ordering of operations during transfer

Merge-1(α , β) \rightarrow

Transfer α :

Reconstruct heads \rightarrow

Reconstruct adjuncts \rightarrow

Reconstruct A/A' -movement \rightarrow

Agree-1 \rightarrow

LF-interface and legibility \rightarrow

Semantic interpretation

The sequence is performed in a one fell sweep, in a reflex-like manner; it is not possible to evaluate the operation only partially or backtrack some moves while executing others. Notice the application of the left branch phase principle in the above example. The fact that the transfer operations must be ordered, or that they have logical priority properties, is nontrivial.

4.9 Lexicon and morphology

4.9.1 From phonology to syntax

Most phonological words enter the system as polymorphemic units. The lexico-morphological component decomposes phonological words into its constituent components. This happens in several stages. The lexicon first matches phonological words with morphological decompositions. A morphological decomposition consists of a linear string of morphemes $m_1\#...\#m_n$. These morphemes, which we assume designate primitive lexical items, then retrieve the corresponding primitive lexical items $M_1 + \dots + M_n$ from the same lexicon. The whole sequence operates with linear strings, not hierarchical or otherwise complex objects. The primitive lexical items are then fed into syntax, where they form complex heads $M_1(M_2...(M_n))$. Head reconstruction, finally, extracts them into head spreads $[M_1...[M_2...[...M_n...]]]$. The whole process is illustrated in **Virhe. Viitteen lähde ei löytynyt..**

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

4.9.2 Lexical items and lexical redundancy rules

Primitive lexical items ($M_1 \dots M_n$) are sets of features. Lexical features emerge from three distinct sources. One source is the *language-specific lexicon*, which stores information specific to lexical items in a particular language. For example, the Finnish sentential negation behaves like an auxiliary, agrees in ϕ -features, and occurs above the finite tense node in Finnish. Its properties differ from the English negation *not*. Some of these properties are so idiosyncratic that they must be part of the language-specific lexicon. One such property could be the fact that the negation selects T as a complement, which must be stated in the language-specific lexicon to prevent the same rule from applying to the English *not*.

Another source of lexical features comes from a set of universal *lexical redundancy rules*. For example, the fact that transitive verbs can select object arguments need not be listed separately in connection with each transitive verb. This fact emerges from a list of universal redundancy rules which are stored in the form of feature implications $F_1 \dots F_n \rightarrow G_1 \dots G_m$ which determine that if a lexical item has features $F_1 \dots F_n$ it will also get feature $G_1 \dots G_m$. When a lexical item is retrieved, its feature content is fetched from the language specific lexicon and processed through the redundancy rules. If there is a conflict, the language-specific lexicon wins. Lexical redundancy rules can define language-specific features when some of the trigger features are language specific, specifying the language that particular lexical item belongs to. We imagine the redundancy rules as forming a primitive mini-grammar or a lexical template that defines the basic properties of lexical items.

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 modified by lexical redundancy rules. Since redundancy rules can be language specific, the constitution of the universal morphemes can depend on the languages.

4.9.3 Derivational and inflectional morphemes

Inflectional and derivational morphemes differ in how they are processed. Derivational morphemes are processed as described above: they are mapped into primitive lexical items and streamed into syntax, where they form first complex heads and the head spreads. Inflectional features are mapped into features which are inserted inside the closest lexical item in the sequence. For example, the verb *admires* is decomposed into three elements V+T+3sg in the lexicon, where

the last element represents the third person agreement features. It is mapped into the corresponding phi-features (singular, third person) which are inserted inside T as lexical features, thus $V+T+3sg \sim T_{3sg}(V) \sim [\dots[T_{3sg} [\dots V\dots]]$. If we specified in the lexicon that 3sg refers to a derivational element, then a separate “agreement head” Agr would be generated instead.

4.9.4 Morphology

Possible morphological decompositions are those morpheme sequences (both derivational and inflectional) which can be used as seeds for legitimate LF-interface objects. Notice that the whole sequence mapping phonological words into head spreads is regulated and cannot produce alternatives or decision points. For example, a logically possible but empirically impossible verb that contains two tense morphemes *admireded* $\sim V+T_{pst}+T_{pst}$ can be generated in the lexicon, and processed by the system, but it will never lead into a legitimate LF-interface object. Morphology, therefore, becomes part of syntax and semantics. Whether this is sufficient has so far not been examined in any study.

4.10 Narrow semantics

4.10.1 Syntax, semantics and cognition: the framework

Semantics is the study of meaning. In this study we construct the notion of meaning in the following way. We assume that linguistic conversation and/or communication projects a set of semantic objects that represents the “things” that the ongoing conversation or communication “is about.” This set or structure contains things like persons, actions, thoughts or propositions as represented by the hearer and the speaker. This temporary semantic repository is called *global discourse inventory*. The discourse inventory can be accessed by global cognition, operations like thinking, decision making, planning, problem solving and others. Linguistic expressions, in turn, are utilized to introduce, remove and update entities in the discourse inventory. To illustrate, consider the sentence *the horse raced past the barn*. We assume that the hearer projects the following entities into the global discourse inventory as s/he processes the sentence.

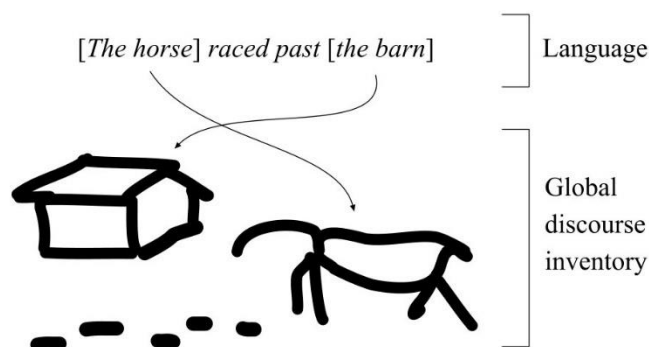


Figure 37. Narrow semantics projects semantic objects into the global discourse inventory.

We assume that this process of projecting and updating entities inside the global discourse inventory gives linguistic expressions their “meaning” in the narrow technical sense relevant to the present work. This of course constitutes a very restricted understanding of what “meaning” is, but it also feasible in the sense that an algorithmic approach able to handle nontrivial semantic data becomes possible.

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*. It can be viewed as a gateway that encapsulates the syntactic pathway and mediates communication in and out. Cognitive systems that are outside of narrow semantics belong to global cognition and incorporate language-external processes like thinking, emotions and conscious decision-making. Narrow semantics is implemented by special-purpose functions or modules which interpret linguistic features arriving through the syntax-semantics interface. We can perhaps imagine language as a cognitive system that grammaticalizes extralinguistic cognitive resources.

Different grammatical features are interpreted by qualitatively different semantic systems. Information structure (notions such as topic and focus) is created by different processing pathway than the system that interprets quantifier scope. Narrow semantics can therefore be also viewed as a gateway or “router,” where the processing of different features is distributed to different language-external systems or where the language faculty or syntactic processing pathway makes contact with other cognitive systems (see Figure 38).

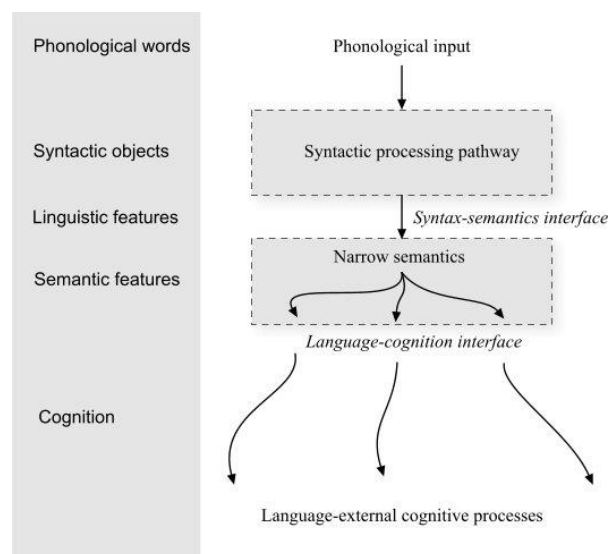


Figure 38. The general cognitive architecture. Narrow semantics bleeds input from language and feeds it in a transformed shape to other cognitive systems with which it makes contact. What these connections are or can depend on the innate neuronal architecture of the human brain.

A proper name such as *John* can be thought of as referring to a simple “thing” like an individual person. A pronoun like *he*, on the other hand, can refer in principle to several objects in the discourse inventory (*John₁ admires Simon₂, and he_{1,2} is very clever*), and the same is true of quantifiers like *every man* or *two men*. Furthermore, expressions like *no one* do not refer to anything, yet they, too, have meaning. Even *John* can be ambiguous if there are several men with the same name. The general problem is that there is nothing in the expression itself that determines unambiguously what it denotes, so the listener must always perform some type of selection and/or guessing. To handle this, all expressions are first linked to unambiguous semantic entries inside narrow semantics, representing their intrinsic semantic properties, and these intermediate representations are transformed into actual denotations that point into semantic objects in the global discourse inventory accessed by other cognitive processes. To illustrate, consider a short conversation *The horse raced past the barn; it was very fast*. The first sentence will establish that there are two things in the global discourse inventory the sentence speaks about: the horse and the barn. The next sentence then makes a claim about some “it” that we must link with something. This pronoun can denote four things in this conversation: the horse, the barn, the whole event, or a third entity not yet mentioned, as shown in Figure 39.

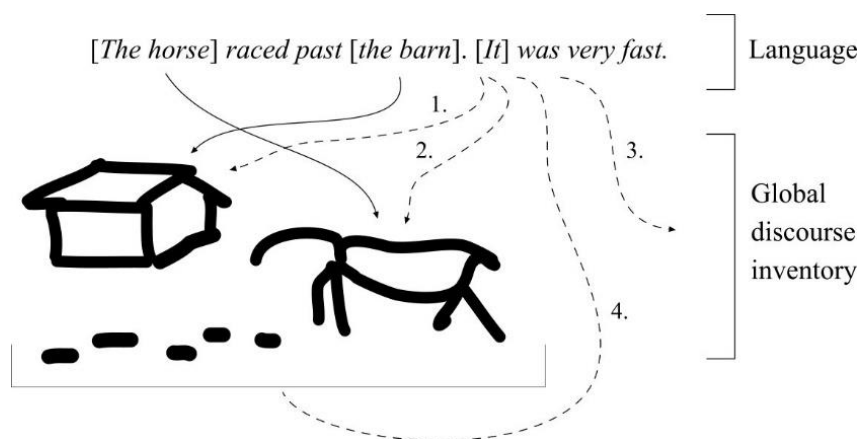


Figure 39. Possible denotations for expression *it*.

Narrow semantics calculates these possible denotations by using the properties of the referring expression itself (e.g., nonhuman, singular, third party in the conversation) and what is contained in the global discourse inventory at the time when the expression is interpreted (the horse, the barn, the event, a possible third entity). The truth-value of the sentence, and thus its ultimate

meaning in a context, is calculated when all referential expressions are provided some denotation. This is called *assignment*. Thus, the most plausible assignment is one in which *the horse* denotes the horse, *the barn* denotes the barn, and *it* denotes the horse as well. The assignment in which *it* denotes the barn is implausible, but possible in principle. The model provides all possible assignments and their rankings as output. Assignments are calculated at the language-cognition interface and not inside the syntactic processing pathway.

When a linguistic feature such as [SINGULAR] is transformed into a formal understood by global cognition, what we mean is that the formal signal representing that feature in the linguistic output will activate a corresponding signal or representation (representing, say, ‘one’) inside the extralinguistic cognitive system. In the case of quantifiers such as *some*, the mapping is more complex but the principle is the same. This quantifier signals that we are supposed to select some (but no matter what and how many) objects from the global discourse inventory. I assume that the operation of ‘selecting some’ is part of the human cognitive repertoire accessed by narrow semantics, and that this is the reason a quantifier (or a lexical feature corresponding to it) can exist. The feature/quantifier gives an instruction for that system to select some arbitrary object(s), perhaps satisfying additional criteria (e.g., *some men* selects ‘some objects that are men’).

In short, we assume that language is used to manipulate the contents of the discourse inventory, which then comprises large part of what we mean by the term “meaning” in this study. The manipulation is handled by connecting the syntactic pathway to the discourse inventory by means of an intermediate system called narrow semantics. Obviously there are other cognitive processes that have access to the discourse inventory as well, such as vision or hearing.

4.10.2 *Incremental semantics*

Semantic interpretation is created incrementally when the input sentence is consumed. For example, as soon the hearer has analyzed and processed an expression *the horse*, the corresponding entity is projected into the global discourse space. This operation takes place during transfer when the expression arrives at the syntax-semantics interface and triggers semantic interpretation.

This type of realistic incremental semantics was part of an earlier version of the model reported in (Brattico 2021a), but is currently replaced by a simulation. Syntactic backtracking requires semantic backtracking in the semantic component, which would be computationally wasteful due to the amount of semantic data structures involved, but also irrelevant, since it is unlikely that semantic backtracking is part of the real cognitive toolkit or a realistic psycholinguistic model. Most of the semantic interpretation is currently created for the final solution arriving at the syntax-

semantics interface; transfers of partial phrase structures (left branches, right adjuncts) are not interpreted.

4.10.3 *Argument structure*

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

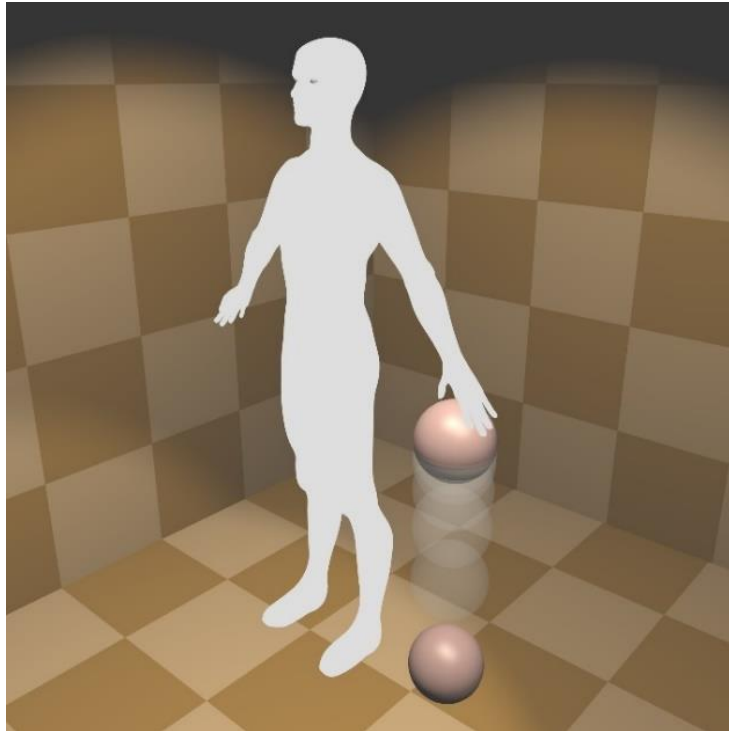


Figure 8. A non-discursive, perceptual or imagined representation of the meaning of the sentence *John dropped the ball*.

Imagine you have an non-discursive (analogous, continuous) experience of a person dropping a ball, depicted in the above figure. We assume that instead of a static figure, you are looking at a dynamic scene that evolves over time. Under normal brain function the visual-conceptual experience will be structured in a certain fairly robust way such 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* obviously correspond or grammaticalize these conceptual objects, in this case John, the ball and the event of dropping expressed by the verb. There is more, however: the meaning emerges not simply from a list of the elements, but also from the structure into which

they have been embedded. A sentence *the ball dropped John* has a vastly different meaning. Lexical items *and their structure* are linked with the contents of the discourse inventory and, ultimately, the everyday human experience.

Let us consider the grammatical representation of the sentence (55) and how the different parts may correlate with the conceptual experience depicted in Figure 8.

(55) [_{TP} John_I [_{TP} T_{pst} [_{VP} _____I [_{VP} v [_{VP} fall [the ball]]]]]]]

The combination of the object the ball and the verb, the lower VP, correspond to a subevent where the ball falls. Thus, a bare verb together with an object is typically interpreted as denoting an event that involve one participant. This is then combined with the small verb *v* and the agent argument John, which is interpreted as the causer of the subevent. That is, John is the agent of the event because he causes the ball to fall. The combination of *fall* + *v* will be the transitive verb *drop* ‘cause to fall’. Consequently, the argument at SpecvP is interpreted as the agent of the event, while the argument inside the VP is interpreted as the patient that must undergo the event caused by the agent. These interpretations are created when the representation arriving at the LF-interface is interpreted semantically by narrow semantics. The parser itself does not see these interpretations, it sees only the selection features; the interpretation is read off from a representation generated by the model at the syntax-semantics interface. The tense *T* adds tense information to the event, but the argument at SpecTP will not get a secondary thematic interpretation; in Finnish it will be interpreted as the topic. In general, though, we assume that the grammatical heads and the structures they create map into entities in the discourse inventory and human experience, in this case objects, events, causation and temporal properties. The elements must be organized in some specific way in order for this interpretation to succeed, and the formal selection restriction features present in the lexicon guide the syntactic processing pathway towards solutions which satisfy these ultimately semantic requirements grounded in the structure of human experience.

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

4.10.4 Antecedents and control

The assumptions specified in the section elucidating Agree-1 (4.8.8) leave room for a situation in which an unvalued phi-feature or a whole phi-set arrives to the LF interface unvalued. Recall that a *predicates* are characterized by the fact that they must be linked with arguments. This behavior is caused by the fact that predicates possess unvalued phi-features. Unvalued phi-features, in turn, may get valued by the operation Agree-1, which locates a local referential argument and copies its phi-features to the predicate. In the case of a sentence *John sleeps*, for example, *sleep* comes with an unvalued phi-feature that is valued by the phi-features of *John*. Consequently, John is interpreted as its argument. Obviously, then, if Agree-1 does not take place, the unvalued phi-features of the predicate may remain unvalued until LF-interface. Finally, Agree-1 may fail to apply if either (1) no local argument is present (56)a-b or (2) the predicate is marked for –VAL which prevents it from agreeing with anything (56)c.

- (56) a. Pekka sanoi että — **haluaa** nukkua.
 Pekka said that want-3sg sleep
 ‘Pekka said that he wants to sleep.’
 b. John wants to **sleep**.
 c. Pekka halusi **nukku-a**.
 Pekka wanted sleep-A/INF (no agreement possible)
 ‘Pekka wanted to sleep.’

When this happens, an unvalued feature or features trigger(s) an *LF-recovery* process that attempts to find a suitable argument by searching for an *antecedent*. LF-recovery is activated inside narrow semantics. An antecedent is located by establishing an upward path (22) from the triggering feature/head to the antecedent. The resulting antecedent relations are illustrated in (57).

- (57) a. Pekka sanoi että — **haluaa**_{φ=Pekka} nukkua.
 Pekka said that want-3sg sleep
 b. John wants to **sleep**_{φ=John}.
 c. Pekka halusi **nukku-a**_{φ=Pekka}.
 Pekka wanted sleep-A/INF
 ‘Pekka wanted to sleep.’

If LF-recovery finds no antecedent, the argument is interpreted as generic, corresponding to ‘one’ (58).

(58) To leave_{φ=‘one’} now would be a big mistake.

LF-recovery constitutes some type of last resort operation which scans the working memory if the predicate still remains without an associated argument at the LF-interface.

4.10.5 *The pragmatic pathway*

In addition to the syntactic processing pathway, there exists a separate pragmatic pathway that monitors the incoming linguistic information and uses it to create pragmatic interpretations for the illocutionary act and/or communicative situations associated with the input sentence. The same pragmatic pathway computes topic and focus properties to the extent that they have not been grammaticalized and/or are based on general psychological characteristics of the communicative situation. This means that what linguistics describe as ‘information structure’ is partitioned into an interpretative extralinguistic pragmatic component and a syntactic (grammaticalized) component.

The pragmatic pathway works by allocating attentional resources to the incoming expressions and the corresponding semantic objects and by notifying if an element is attended in an unexpected (‘too early’ versus ‘too late’) position. The current implementation relies on two syntax-pragmatics interfaces to handle these cases. The first interface occurs very early in the processing pipeline – at the lexical stream currently – and registers all incoming referential expressions and allocates attentional resources to them. Another interface is connected to the component implementing discourse-configurational word order variations during transfer. It responds to situations in which some expression occurs in a noncanonical ‘too early’ or ‘too late’ position, and then generates the corresponding pragmatic interpretations ‘more topical’ and ‘more focus-like’, respectively. The results are shown in the field “information structure” in the output. The idea is that 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 lexical *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.

4.10.6 *Operators*

Operators are processed inside their own 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 for the operator-variable module for interpretation. If the original

operator feature is [OP:F], then in most cases the scope marker is closest lexical head inside the path with [OP:F][FIN]. The finite operator cluster [OP:F][FIN] usually emerges during reconstruction.

4.10.7 Binding

All referential expressions such as pronouns, anaphors or proper names must be linked with some object or objects in the semantic inventory so that both the hearer and speaker know what is “talked about.” This is a nontrivial problem for language comprehension, because all such expressions are ambiguous. Even a proper name such as *John* can refer to any male person in the current conversation whose name is or is assumed to be John. The model restricts possible denotations by using whatever lexical features are available in the lexical items themselves and whatever is available in the global discourse inventory. For example, a proper name *John* can only denote a single male person what that name. In the same way, *she* cannot denote a male person. Neither can denote something that does not (yet) exist in the global discourse inventory. Some referential expressions also impose structure-dependent restrictions on what they can denote. Reflexives like *himself* must be coreferential with a nonlocal antecedent (59), *him* cannot denote a too local antecedent (60), and R-expressions such as proper names must remain free (61).

- (59) a. John₁ admires himself_{1,*2}.
 b. *John’s₁ sister admires himself₁
- (60) a. John₁ admires him_{*1,2}
 b. John’s₁ sister admires him_{1,2}
- (61) a. He₁ admires John_{*1,2}.
 b. His_{1,2} sister hates John₁.

Binding theory (Chomsky 1981) 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. One possibility is that these restrictions are determined in the syntactic component and the information is then forwarded to narrow semantics. Another is that they are computed only in the latter component. The model assumes the latter model, in other words that binding conditions themselves operate at the interface (Brattico 2021b). Specifically, it is assumed that nominal expressions (anaphora, pronouns, R-expressions) contain grammaticalized features that provide “instructions” for a cognitive system that then filters possible assignments as shown in the data above.

The system can be described nontechnically as follows. Each assignment constructed by narrow semantics is weighted in terms of plausibility. Several factors affects these weights, including

contextual pragmatic plausibility. The weighting function can also be controlled by instructions that can be grammaticalized in language. Some of these grammaticalized instructions generate strict and deterministic restrictions seen in the binding conditions above. They operate by using the syntax-semantics interface objects to generate a reference set of objects that are included or excluded from assignment when considering the denotation for some particular expression α . This results in the limitations exhibited by the data above. The fact that the binding conditions are structure-dependent follows from the assumption that they are computed on the basis of LF interface objects that the narrow semantics has access to.

5 Performance

5.1 Human and engineering parsers

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

5.2 Mapping between the algorithm and brain

Figure 32 maps the components of the model into their approximate locations in the brain on the basis of neuroimaging and neurolinguistic data.

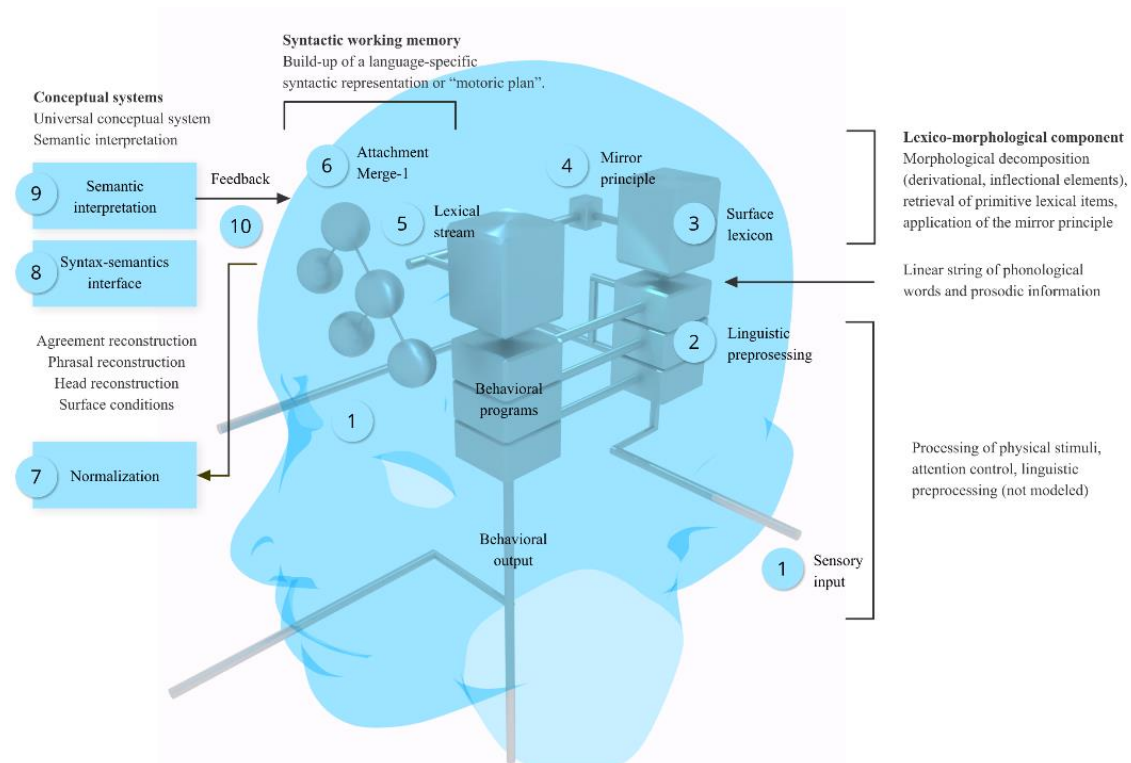


Figure 32. Components of the model and their approximate locations in the human brain. See the main text for explanation.

Sensory stimulus (1) is processed through multiple layers of lower-level systems responsible for attention control and modality-specific filtering, in which the linguistic stimuli is separated from other modalities and background noise, localized into a source, and ultimately presented as a linear string of phonological words (2). The current model assumes a tokenized string (2) as input. Brain imaging suggests that the processing of auditory linguistic material takes place in and around the superior temporal gyrus (STG), with further processing activating a posterior gradient towards the Wernicke's area that seems responsible for activating lexical items (3)(Section 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)(Section 4.1). It is possible that these same regions implement transfer (Section 4.8), as damage to Broca's region seem to affect transformational aspects of language comprehension. The syntax-semantic interface (LF-interface) is therefore quite conceivably also implemented within the anterior regions and can be assumed to represent the endpoint of linguistic processing. There is very little neurolinguistic data on what happens after that point. The architecture assumes that the endpoint of syntactic processing makes contact with other cognitive systems, possibly anywhere in the brain, via narrow semantics.

5.3 Cognitive parsing principles

5.3.1 *Incrementality*

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

There is evidence that the human language comprehension system is incremental. It is possible to trick the system into making a wrong decision on the basis of incomplete local information, which shows that the parser does not wait for the appearance of further words before making its decisions. This can be seen from (62), in which the parser interprets the word *raced* as a finite verb despite the fact that the last word of the sentence cannot be integrated into the resulting structure.

(62)

- a. The horse raced past the barn.
- b. The horse raced past the barn fell.

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

```
1   the

the + horse

2   [the horse]

[the horse] + T

3   [[the horse] T]

[[the horse] T] + race

4   [[the horse] T(V)]

[[the horse] T(V)] + past

5   [[the horse] [T(V) past]]

[[the horse] [T(V) past]] + the

6   [[the horse] [T(V) [past the]]]

[[the horse] [T(V) [past the]]] + barn

7   [[the horse] [T(V) [past [the barn]]]]

[[the horse] [T(V) [past [the barn]]]] + T

8   [[the horse] [T(V) [past [[the barn] T]]]]

[[the horse] [T(V) [past [[the barn] T]]]] + fell

9   [[the horse] [T(V) [past [the [barn T]]]]]

[[the horse] [T(V) [past [the [barn T]]]]] + fell

10  [[the horse] [T(V) [[past [the barn]] T]]]

[[the horse] [T(V) [[past [the barn]] T]]] + fell

11  [[[the horse]:11 [T [__]:11 [race [past [the barn]]]]]] T]
```

[[[[the horse]:11 [T [__:11 [race [past [the barn]]]]]] T] + fell
 12 [[[[the horse]:12 [T [__:12 race]]] past]
 [[[[the horse]:12 [T [__:12 race]]] past] + the
 13 [[[[the horse]:12 [T [__:12 race]]] [past the]]
 [[[[the horse]:12 [T [__:12 race]]] [past the]] + barn
 14 [[[[the horse]:12 [T [__:12 race]]] [past [the barn]]]
 [[[[the horse]:12 [T [__:12 race]]] [past [the barn]]] + T
 15 [[[[the horse]:12 [T [__:12 race]]] [past [[the barn] T]]]
 [[[[the horse]:12 [T [__:12 race]]] [past [[the barn] T]]] + fell
 16 [[[[the horse]:12 [T [__:12 race]]] [past [the [barn T]]]]
 [[[[the horse]:12 [T [__:12 race]]] [past [the [barn T]]]] + fell
 17 [[[[the horse]:12 [T [__:12 race]]] [[past [the barn]] T]]
 [[[[the horse]:12 [T [__:12 race]]] [[past [the barn]] T]] + fell
 18 [[[[the horse]:12 [T [__:12 race]]] <past [the barn]>] T]
 [[[[the horse]:12 [T [__:12 race]]] <past [the barn]>] T] + fell
 19 [the [horse T]]
 [the [horse T]] + race
 20 [the [horse T(V)]]
 [the [horse T(V)]] + past
 21 [the [horse [T(V) past]]]
 [the [horse [T(V) past]]] + the
 22 [the [horse [T(V) [past the]]]
 [the [horse [T(V) [past the]]] + barn
 23 [the [horse [T(V) [past [the barn]]]]]
 [the [horse [T(V) [past [the barn]]]]] + T
 24 [the [horse [T(V) [past [[the barn] T]]]]]
 [the [horse [T(V) [past [[the barn] T]]]]] + fell

25 [the [horse [T(V) [past [the [barn T]]]]]]
 [the [horse [T(V) [past [the [barn T]]]]]] + fell

26 [the [horse [T(V) [[past [the barn]] T]]]]
 [the [horse [T(V) [[past [the barn]] T]]]] + fell

27 [[the [horse [T [race [past [the barn]]]]]] T]
 [[the [horse [T [race [past [the barn]]]]]] T] + fell

28 [[the [horse [T [race [past the]]]]] barn]
 [[the [horse [T [race [past the]]]]] barn] + T

29 [[the [horse [T [race [past the]]]]] [barn T]]
 [[the [horse [T [race [past the]]]]] [barn T]] + fell

30 [[the [horse [T [race past]]]] the]
 [[the [horse [T [race past]]]] the] + barn

31 [[the [horse [T [race past]]]] [the barn]]
 [[the [horse [T [race past]]]] [the barn]] + T

32 [[the [horse [T [race past]]]] [[the barn] T]]
 [[the [horse [T [race past]]]] [[the barn] T]] + fell

33 [[the [horse [T [race past]]]] [the [barn T]]]
 [[the [horse [T [race past]]]] [the [barn T]]] + fell

34 [[the [horse [T race]]] past]
 [[the [horse [T race]]] past] + the

35 [[the [horse [T race]]] [past the]]
 [[the [horse [T race]]] [past the]] + barn

36 [[the [horse [T race]]] [past [the barn]]]
 [[the [horse [T race]]] [past [the barn]]] + T

37 [[the [horse [T race]]] [past [[the barn] T]]]
 [[the [horse [T race]]] [past [[the barn] T]]] + fell

38 [[the [horse [T race]]] [past [the [barn T]]]]

```

[[the [horse [T race]]] [past [the [barn T]]]] + fell

39  [[the [horse [T race]]] [[past [the barn]] T]]

    [[the [horse [T race]]] [[past [the barn]] T]] + fell

40  [the horse]

    [the horse] + T/prt

41  [the [horse T/prt]]

    [the [horse T/prt]] + race

42  [the [horse T/prt(V)]]

    [the [horse T/prt(V)]] + past

43  [the [horse [T/prt(V) past]]]

    [the [horse [T/prt(V) past]]] + the

44  [the [horse [T/prt(V) [past the]]]]

    [the [horse [T/prt(V) [past the]]]] + barn

45  [the [horse [T/prt(V) [past [the barn]]]]]

    [the [horse [T/prt(V) [past [the barn]]]]] + T

46  [the [horse [T/prt(V) [past [[the barn] T]]]]]

    [the [horse [T/prt(V) [past [[the barn] T]]]]] + fell

47  [the [horse [T/prt(V) [past [the [barn T]]]]]]

    [the [horse [T/prt(V) [past [the [barn T]]]]]] + fell

48  [the [horse [T/prt(V) [[past [the barn]] T]]]]

    [the [horse [T/prt(V) [[past [the barn]] T]]]] + fell

49  [the [horse [[T/prt [race [past [the barn]]]] T]]]

    [the [horse [[T/prt [race [past [the barn]]]] T]]] + fell

50  [the [horse [[T/prt [race [past [the barn]]]] [T fell]]]] (<= accepted)

```

The exact derivation is sensitive to the actual parsing principles that are activated before the simulation trial. For example, if we assume that the participle verb is activated before the finite

verb (against experimental data), then the model will reach the correct solution without any garden paths.

I do not assume that the backtracking operation visible in the example above is entirely realistic from the psycholinguistic point of view. There are two reasons why it exists. One is that by performing systematic backtracking we let the model to explore the complete parsing tree. If we only generated the first solution, spurious secondary solutions might escape our attention. It is not untypical that the model finds ungrammatical and/or wrong secondary solutions, revealing that it is using a wrong competence model. The second reason is that backtracking gives us important information concerning the model's performance. We can compare the amount of computational resources spend in processing different types of constructions and/or different algorithmic solutions. In addition, realistic language comprehension by the human brain in connection with canonical sentences is seldom subject to any garden pathing, so we can at least aim for a model that finds the correct solution immediately, at least in those cases. Finally, it is possible, in fact a worthwhile goal, to add a realistic backtracking model on the top of the systematic backtracking as an optional component that can be turned on and off as needed. I suspect that real speakers solve garden path problems by starting the parsing from the beginning with additional "noise" added to the decision mechanism. These questions must be explored by psycholinguistic experimentation.

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

5.3.2 *Connectness*

Connectness refers to the property that all incoming linguistic material is attached to one phrase structure that connects everything together. There never occurs a situation where the syntactic working memory holds two or more phrase structures that are not connected to each other by means of some grammatical dependency. It is important to keep in mind, though, that 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 (Section 4.6). We can imagine of them being "pulled out" from the computational pipeline processing the host structure and being processed by an independent

computational sequence. Each adjunct, and more generally phase, is transferred independently and enters the LF-interface as an independent object.

5.3.3 Seriality

Seriality refers to the property that all operations of the parser are executed in a well-defined linear sequence. Although this is literally true of the algorithm itself, the detailed serial algorithmic implementation cannot be mapped directly into a cognitive theory for several reasons. One reason is that each linguistic computational operation performed during processing is associated with a “predicted cognitive cost,” measured in milliseconds, and it is the linear sum of these costs that provides the user the predicted cognitive processing time for each word and sentence; CPU time is ignored. The current parsing model is serial in the sense that the predicted cognitive cost is computed in this way by adding the cognitive 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. The underlying implementation would still remain serial. Another complicating factor is that in some cases the implementation order does not seem to matter. We could simply *assume* that the processing is implemented by utilizing parallel processes. Despite these concerns, it is clear that most of the computational operations implemented by the linear phase model must be executed in a specific order in order to derive empirically correct results. Therefore, for the most part the model assumes that language processing is serial.

5.3.4 Locality preference

Locality preference is a heuristic principle of the human language comprehension which requires that local attachment solutions are preferred over nonlocal ones. A local attachment solution means the lowest right edge in the existing partial phrase structure representation. Thus, the preposition phrase *with a telescope* in (63) is first attached to the lowest possible solution, and only if that solution fails, to the nonlocal node.

(63) John saw the girl with a telescope.

It is possible to run the parsing model with five different locality preference algorithms. They are as follows: *bottom-up*, in which the possible attachment nodes are ordered bottom-up; *top-down*, in which they are ordered in the opposite direction (‘anti-locality preference’); *random*, in which the attachment order is completely random; *Z*, in which the order is bottom first, top second, then the rest in a bottom-up order; *sling*, which begins from the bottom node, then tries the top node and explores the rest in a top-down manner. The top-down and random algorithm constitute baseline controls that can be used to evaluate the efficiency of more realistic principles. The

selected algorithm is defined for each independent study (Section 6.3.4). If no choice is provided, bottom-up algorithm is used by default.

5.3.5 *Lexical anticipation*

Lexical anticipation refers to parsing decisions that are made on the basis of lexical features. The linear phase parser uses several lexical features (Sections 4.3, 6.3.2). The system works by allowing each lexical feature to vote each attachment site either positively or negatively, and the sum of the votes will be used to order the attachment sites. The weights, which can be zero, can be determined as study parameters (Section 6.3.4). This allows the researcher to determine the relative importance of various lexical features and, if required, knock them out completely (weight = 0). Large scale simulations have shown that lexical anticipation by both head-complement selection and head-specifier selection increases the efficiency of the algorithm considerably.

5.3.6 *Left branch filter*

The left branch filter closes parsing paths when the left branch constitutes an unrepairable fragment. The principle operates before other ranking principle are applied. The left branch filter can be turned on and off for each study (Section 6.3.4).

5.3.7 *Working memory*

There is substantial psycholinguistic literature that the operation of the human language comprehension module is restricted by a working memory bottleneck. After considerable amount of simulation and exploration of other models I concluded that this hypothesis must be correct. The user can activate the working memory or knock it off by changing the study parameters in a manner explained in Section 6.3.4 and in this way see what its effects are.

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

does not apply to such constituents, hence the system loses some ‘grammatical intelligence’ when it has to re-evaluate wrong parsing decisions made earlier.

5.3.8 *Conflict resolution and weighting*

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

5.4 Measuring predicted cognitive cost of processing

Processing of words and whole sentences is associated with a predicted cognitive cost, measured in milliseconds. This is done by associating each word with a preprocessing time depending on its phonetic length (currently 25ms per phoneme) and then summing predictive cognitive costs from each computational operation together. Most operations are currently set to consume 5ms, but the user can define these in a way that best agrees with experimental and neurobiological data. Reactivation of a constituent that is not inside the active working memory consumes 500ms. The resulting timing information will be visible in the log files and in the resource outputs. The processing time consumed by each sentence is simply the sum of the processing time of all of its words. A useful metric in assessing the relative processing difficulty of any given sentence is to calculate the mean predicted cognitive processing time per each word (total time / number of words). This metric takes sentence length into account. Resource consumption is summarized in the resource output file (Section 6.4.6) that lists each sentence together with the number of all computational operations (e.g., Merge, Agree, Move) consumed from the reading of the first word to the outputting of the first legible solution. The file uses CSV format and can be read into an analysis program (Excel, SPSS, Matlab) or processed by using external Python libraries such as pandas or NumPy.

5.5 A note on implementation

Most of the performance properties are implemented in their own module `plausibility_metrics.py` which in essence determines how the attachment solutions (Section 4.1) are filtered and ordered. The abstract linear parser, which does not implement any performance properties by itself, sends all available attachment solutions to this module, which will first focus the operation to those nodes which are in the active working memory; the rest are not processed. The active nodes are then filtered, so that only valid solutions remain. The user can knock off all filters by changing the parameters of the study. The remaining nodes are then ordered by applying the selected locality preference algorithm, which provides the default ordering, and then by applying all other ranking principles such as lexical anticipation. Finally, the concatenated list of ordered nodes + nodes not active in the working memory and not processed are returned. The parser, which receives this list from the plausibility module, then uses this order in organizing its parsing derivation. Notice that the inactive nodes that are not in the active working memory must be part of the list as the parser must be able to explore them if everything else fails, but since the plausibility module does not process them, their order is independent of the incoming word and the partial phrase structure representation currently being constructed.

6 Inputs and outputs

6.1 Installation and use

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

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

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

```
python lpparse
```

into the command prompt, which will parse all the sentences from the designated test corpus file in a study folder. This command will call the Python interpreter, as specified in the path-variable, and then feeds the `__main__.py` module from the folder `/lpparse` into it. This will then call any other module, as required.

Each trial run that is launched by the above command involves a host of internal parameters that the user can configure. These involve things such as where is the lexicon, test corpus, what heuristic principles should be used, and many others. The information can be provided in several ways. One way is to provide it inside a configuration file called `config_study.txt`. If the parser is launched without any parameters, as in the example above, then it will try to find this file from the installation directory. Usually this file is present in any version currently being developed. If the file is not found, however, default values will be used. Finally, any parameter that is defined in the `config_study.txt` can also be defined as an input parameter to the program, which will

overwrite any specifications found from the configuration file. This makes it possible to control the execution of the script from an external source, say from an external program that one might want to use to organize scripts that perform several studies. Figure 35 shows the contents of the configuration file I have currently on my computer; explanations follow.

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

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

Each line has two fields: the key and a value, separated by semicolon. The key determines the name of the parameter. For example, the key `test_corpus_folder` determines the name of the parameter that defines the folder from where the program tries to find the test corpus file. It is followed by the name of the folder. Each parameter is read in the same way. Whatever keys and values are provided will be read into a settings data structure (dictionary) that can then be used anywhere in the program. It is not a requirement that the user specifies all keys in this file. If a key or parameter is missing, then that parameter is not used, or a negative value is assumed. If a parameter is missing that is mandatory for normal operation, such as the test corpus file, the program will attempt to use a default value. If also that strategy fails, then an error message will likely appear. Keys cannot contain white spaces, values can.

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

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

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

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

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

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

6.2 General organization

The model was implemented and formalized as a Python 3.x program. It contains three main components. When the user launches the program, module `__main__.py` is first run. This module takes care of reading and interpreting the command line parameters, and it can be used to diverge the execution to different modules based on command line parameters. The first component that belongs to the model itself is the main script `main.py` responsible for running one study. It reads an input corpus containing test sentences and other input files, such as those containing lexical information, prepares the parser (with some language and/or other environmental variables), runs the test corpus with the parser, and processes and stores the results. The architecture is illustrated in Figure 12. The code for the main script is explained in Section 6.5.

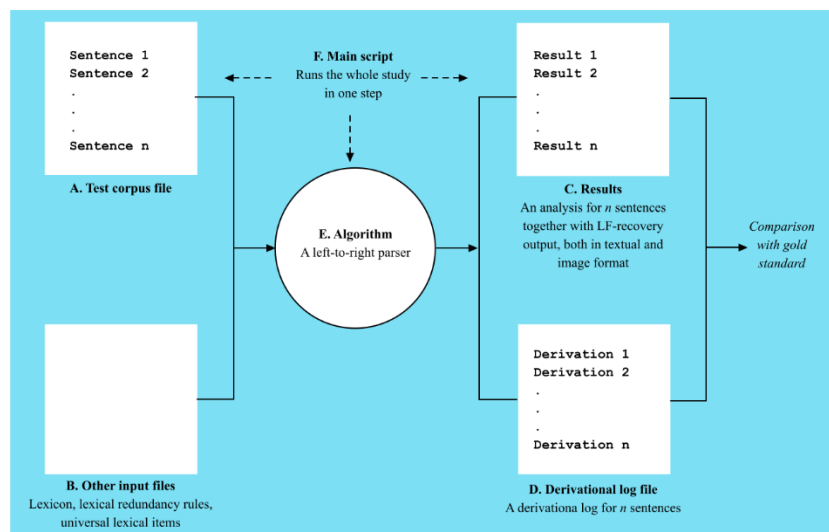


Figure 12. Relationships between the input, main script, linear phase parser and the output.

Only two output files are shown, the main results file and the derivational log file.

Any complete study is run by launching the main script once, which provides a mapping between the input files and output files, and which are stored as raw data associated with each study. The user cannot interfere the execution. The details of the operation of the main script will be elucidated further below.

The second component is the language comprehension module, which receives one sentence as input and produces a set of phrase structures and semantic interpretations as output. That component contains the empirical theory. Finally, the program contains support functions, such as logging, printing, and formatting of the results, reporting of various program-internal matters, and others. These are not part of the empirical theory.

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

Table. An alphabetical list of individual program modules

MODULE	DESCRIPTION
adjunct_constructor.py	This module processes externalization, in which a given element and/or the surrounding phrase is externalized, i.e. moved to the secondary system for independent processing. Linguistically it corresponds to the situation in which the element is promoted into an adjunct. It makes decisions concerning the amount of surrounding structure that will be externalized.
adjunct_reconstruction.py	Adjunct reconstruction takes place during transfer. It detects misplaced adjoinable phrases and reconstructs them by using tail features. It uses adjunct_constructor.py when needed.
agreement_reconstruction.py	Performs agreement reconstruction (Agree-1) for heads with +VAL.
diagnostics.py	Performs statistical analyses of the raw output data.
extraposition.py	Contains code that is used in extraposition, which refers to a type of externalization. Extraposition is attempted during transfer at two stages, first after head reconstruction and then as a last resort. In both cases, some fragment of the structure is externalized. During the first sweep, the system reacts to ungrammatical (and hence unrepairable) selection between reconstructed heads.
feature_disambiguation.py	This module performs feature manipulation (feature inheritance in the standard theory). It is currently solving the ambiguity with ?ARG feature and does

	nothing else, i.e. it sets the feature ?ARG into +ARG or -ARG. This is relevant for control. It is possible that its effects should be explained by lexical ambiguity.
global_cognition.py	Handles all processes and representations that are part of extralinguistic global cognition.
knockouts.py	Contains metafunctions which allow the user to parametrize the model, i.e., to knockout various components of the algorithm.
head_reconstruction.py	This module contains the code taking care of head reconstruction during transfer.
language_guesser.py	Hosts the code which determines the language used in an input sentence. The constructor will first read the lexicon and extract the languages available there, based on LANG features. The guesser will then determine the language of an input sentence on the basis of its words.
lexical_interface.py	This module reads and processes lexical information. Lexical information is read from the three external files and further processed through a function that applies “parameters”. It is stored into a dictionary. Each parser object has its own lexicon that are initialized for each language in the main script.
lexical_stream.py	Defines properties characterizing the lexical stream which “streams” primitive lexical items and inflectional features from the lexico-morphological component into the syntactic component.
LF.py	Processes LF-interface (syntax-semantics interface) objects, with the main role being the checking of LF-legibility. It also hosts LFmerge operations, which are used in the production side to generate representations.
linear_phase_parser.py	This module defines the parser and its operations (main parse function plus the recursive function called by the former). This is a purely performance module: it reads the input sentence, generates the recursive parse tree, evaluates and stores the results.
log_functions.py	Contains two logging related operations, one which logs the sentence before parsing and another which stores the results.
local_file_system.py	Handles all I/O behaviors, including console.
narrow_semantics.py	A gateway or “shell” that wraps the syntactic pathway and sends grammatical (grammaticalized) features to various subsystems for interpretation. These interpretations then generate objects into the discourse inventory where they become visible for global cognition.
main.py	Function that is executed when the user launches the program from the command prompt. This function interprets command line arguments and prepares the study accordingly. It runs one simulation study.
morphology.py	Contains code handling morphological processing, such as morphological decomposition and application of the mirror principle. This module uses only linear representations.
multistudy.py	List of functions that allows one to run several studies at once. Does not work currently and will not be needed in the future.
parse.py	Main script. This script is written as a linear sequence of commands that prepare the parsers (for each language), sends all input sentences into the appropriate parser and stores the results.

phrasal_reconstruction.py	Contains code implementing phrasal reconstruction, both A-bar reconstruction and A-reconstruction. These operations are part of transfer.
phrase_structure.py	A class that defines the phrase structure objects and the grammatical configurations and relations defined on them.
SEM_LF-recovery	Performs LF-recovery and is called by narrow semantics.
SEM_operators_variables	Interprets operator-variable constructions and populates the discourse inventory accordingly.
SEM_pragmatic_pathway	Handles computations involved in the pragmatic pathway that currently computes information structure and grammaticalized discourse features [D:].
SEM_predicates_relations_events.py	Handles computations involved in the semantics of predicates, relations and events.
SEM_quantifiers_numerals_denotations.py	Handles the interpretation of ordinary referential expressions such as quantifiers, numerals and other referential arguments, including pronouns, anaphora and R-expressions.
support.py	Contains various support functions that are irrelevant to the empirical model itself.
surface_conditions.py	The module contains filters that are applied to the spellout structure. In the current implementation it only contains tests for incorporation integrity that are used in connection with clitic processing. It will contain also functions pertaining to surface scope.
transfer.py	This module performs the transfer operation. It contains a list of subprocesses in a specific order of execution (head reconstruction, feature processing, extraposition, adjunct reconstruction, phrasal reconstruction, agreement reconstruction, last resort extraposition)
visualizer.py	Hosts the code used to generate images of phrase structure trees.

Individual studies are associated with specific input files inside the /language data working directory subfolder, which contains a further subfolder for each study, published, submitted or in preparation. A copy of each lexical file exists also in the language data working directly, which makes it possible to work with one “master lexicon.” Once a study is published, however, a copy of the lexical resources used in that study should be stored in connection with the rest of the study-specific materials inside the specific folder.

Sometimes a single study involves running several complete trials that should be stored. These are separated from each other by using letters. Thus, names such as *study-1_a* and *study-1_b* refer to two complete simulations (*a*, *b*) that have been executed inside the same study. The user should always use the latest simulation, based on alphabetical ordering. Thus, in this example the user should use the data stored in the folder *study-1_b*.

6.3 Structure of the input files

6.3.1 Test corpus file (any name)

The test corpus file name and location are provided in the `config_study.txt`. Certain conventions must be followed when preparing the file. Each sentence is a linear list of phonological words, separated by white space from each other and following by next line (return, or `\n`), which ends the sentence. Words that appear in the input sentences must be found from the lexicon exactly in the form they are provided in the input file, which means that the user must normalize the input. For example, do not use several forms (e.g. *admire* vs. *Admire*) for the same word, do not end the sentence with punctuation, and so on. Depending on the research agenda you might want to consider using a fully disambiguated lexicon.

Special symbols are used to render to output more readable and to help testing. Symbols `#` and 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, which is useful if they belong to some other language than English. Symbol `&` is also read as a comment, but it will appear in the results file as well. This allows the user to leave comments into the results file that would otherwise get populated with raw data only. Should the user want to group the sentences by using numerical coding, this is possible by writing `=>x.y.z.a`, for example `=>1.1.1.0`. This will label all following sentences with that numerical code, until another similar line occurs. These numbers are very useful if we want later to analyze the results on the basis of some grouping scheme. If a line is prefaced with `%`, the main script will process only that sentence. This functionality is used if 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. The rest of the sentences are then ignored. Command `=STOP=` at the beginning of a line will cause the processing to stop at that point, allowing the user to process only *n* first sentences. To being processing in the middle of the file, use the symbol `=START=` (in effect, sentences between `=START=` and `=STOP=` still be processed).¹² Figure 14 is a screen capture from one test corpus file to illustrate what it looks like.

¹² It is possible to use several `=START=` and `=STOP=` commands. They are interpreted so that all previous items are disregarded each time `=START=` is encountered, whereas `=STOP=` disregards anything that follows. Thus, only the last `=START=` and the first `=STOP=` will have an effect.

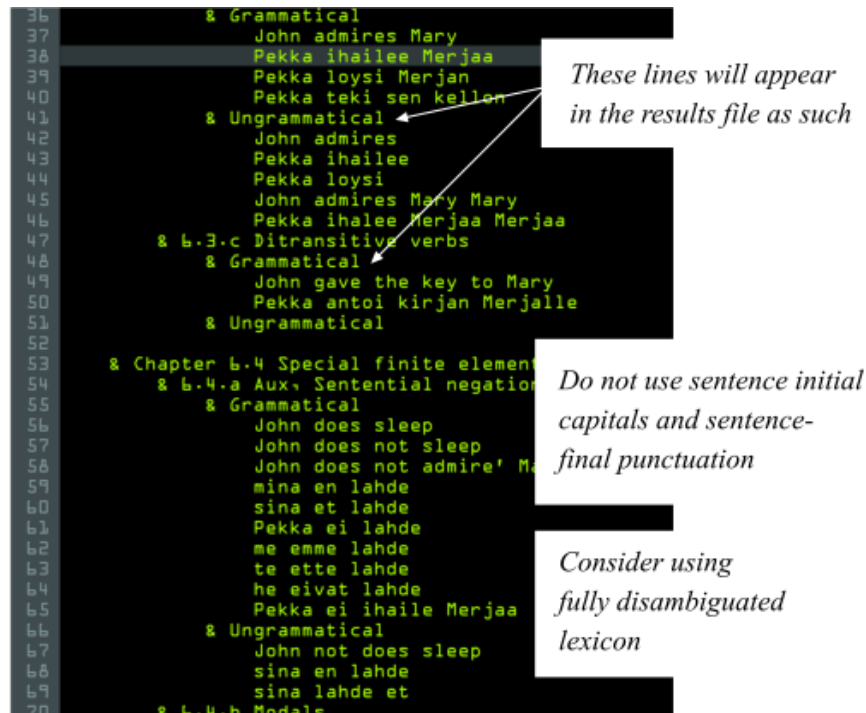


Figure 14. Screen capture of a test corpus file.

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 (64).

- (64) a. John₁ met Mary₂;
 b. He_{1,3} admires her_{2,4}.

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

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

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

```

5  admire  :: admire-#v#T/fin#[E-o] LANG:EN
6  admire' :: admire-#v LANG:EN
7  admires :: admire-#v#T/fin#[E-s] LANG:EN
8  admire- :: PF:admire LF:admire V CLASS:TR -SPEC:Neg -COMP:Neg COMP:D
9
10 adoro   :: adora-#v#T/fin#[E-o] LANG:IT
11 adori   :: adora-#v#T/fin#[E-i] LANG:IT
12 adora   :: adora-#v#T/fin#[E-a] LANG:IT
13 adoriamo :: adora-#v#T/fin#[E-iamo] LANG:IT
14 adorate :: adora-#v#T/fin#[E-te] LANG:IT
15 adorano :: adora-#v#T/fin#[E-no] LANG:IT
16 adora-  :: PF:adora LF:admire V COMP:D LANG:IT
17
18 anta-   :: PF:antaa LF:give V CLASS:DITR -COMP:FIN +SEM:directional LANG:FI
19 antoi   :: anta-#v#T/fin#[E-v] LANG:FI
20
21 asks    :: ask-#v#T/fin#[E-s] LANG:EN
22 ask'    :: PF:ask LF:ask V SPEC:D COMP:D SEM:internal LANG:EN
23 ask-    :: PF:ask LF:ask V SPEC:D COMP:D SEM:internal LANG:EN
24
25 avain_0acc :: avain-#D#[E-0_acc]
26 avain_nom  :: avain-#D#[E-0_nom]
27 avain      :: avain-#D#[E-nom]
28 avaimen_acc :: avain-#D#[E-n_acc]
29 avaimen    :: avain-#D#[E-n_acc]
30 avaimet    :: avain-#D#[E-t_acc]#pl
31 avain-     :: PF:avain LF:key N LANG:FI -SEM:directional
32
33 auton     :: auto-#D#[E-n_acc] LANG:FI
34 auto      :: auto-#D LANG:FI
35 auto-     :: PF:auto LF:car N LANG:FI -SEM:directional
36
37 city      :: PF:city LF:city N LANG:EN
38
39 detesto   :: detest-#v#T/fin#[E-o] LANG:IT

```

Morphological decomposition

List of features associated with a primitive item

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

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

A lexical feature is a string, a “formal pattern,” ultimately a neuronal activation pattern. They do not have further structure and are processed by first-order Markovian operations. The way any given feature reacts inside syntax and semantics is defined by the computations that process these lexical items and the “patterns” in them.¹³ As can be seen from the above screen capture, most features have a ‘type:value’ structure, where the type dictates the system that processes it, value

¹³ They are currently implemented by simple string operations, but in some later iteration all such processing will be replaced by regex processing.

is the input that will generate a specific interpretation. Figure 16 contains a summary of the most important lexical features that the syntax and semantics reacts in the current model. The user can introduce any kind of features (strings) into the lexical elements.

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

Figure 16. Some lexical features

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

6.3.3 Lexical redundancy rules

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

```

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

```

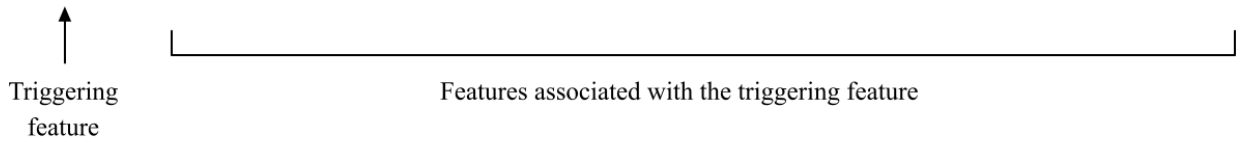


Figure 17. Lexical redundancy rules, as a screen capture from the *redundancy_rules.txt* file.

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

The lexical resources are processed so that the language-specific sets are created first, followed by the application of the lexical redundancy rules. If a lexical redundancy rule conflicts with a language-specific lexical feature, the latter will override the former. Thus, lexical redundancy rules define “default” features associated with any given triggering feature. It is also possible (and in some cases needed) to use language specific redundancy rules. These are represented by pairing the antecedent feature with a language feature (e.g., LANG:FI).

6.3.4 Study parameters (*config_study.txt*)

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

6.4 Structure of the output files

6.4.1 Results

The name and location of the results output file is determined when configuring the main script, either in the external file *config_study.txt* or by arguments when using the multi study functionality. The default name is made up by combining the test corpus name together with “_results”. Each time the main script is run, the default results file is overridden. Once you get

results that are plausible, it is useful to rename the results file to save it and compare with new output. The file begins with time stamps together with locations of the input files, followed by a grammatical analysis and other information concerning each example in the test corpus, with each provided with a numeral identifier. What type of information is visible depends on the aims of the study. The example in Figure 19 shows one grammatical analysis (line 12) together with semantic interpretation (lines 14-24), contents of the global discourse inventory (lines 26-28, in simplified format) and performance metrics (lines 30-35).

10	1. John admires Mary	Input sentence
11		
12	[[D John]:1 [T [__]:1 [v [admire [D Mary]]]]]]	Syntactic analysis
13		
14	Semantics:	
15	Recovery: ['Agent of T(John)', 'Agent of v(John)', 'Patient of admire(Mary)']	
16	Aspect: []	
17	DIS-features: []	
18	Operator bindings: []	Aspects of semantic interpretation
19	Semantic space:	
20	Speaker attitude: []	
21	Assignments:	
22	[D John] ~ 2, [D Mary] ~ 8, Weight 1	
23	Information structure: {'Marked topics': [], 'Neutral gradient': ['[D John]', '[D Mary]'], 'Marked focus': []}	
24	D-features: []	
25		
26	Discourse inventory:	
27	Object 2 in GLOBAL: [D John]	Discourse inventory (semantic objects)
28	Object 8 in GLOBAL: [D Mary]	
29		
30	Resources:	
31	Total Time:1135, Garden Paths:0, Memory Reactivation:0, Steps:3, Merge:6, Move Head:6, Move Phrase:2,	
32	A-Move Phrase:2, A-bar Move Phrase:0, Move Adjunct:0, Agree:2, Phi:5, Transfer:3, Item streamed into syntax:7,	Performance metrics
33	Feature Processing:0, Extraposition:0, Inflection:12, Failed Tense:3,	
34	LF test:5, Filter solution:5, Rank solution:2, Lexical retrieval:19, Morphological decomposition:3,	
35	Mean time per word:378, Asymmetric Merge:36, Sink:9, External Tail Test:13,	
36		

Figure 19. Screen capture from a results file.

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

6.4.2 The log file

The derivational log file, created by default by adding “_log” into the name of the test corpus file, contains a more detailed report of the computational steps consumed in processing each sentence in the test corpus and of the semantic interpretation. The log file uses the same numerical identifiers as the results file. In order to locate the derivation for sentence number 1, for example, you would search for string “# 1” from the log file. What type of information is reported in the log file can be of course decided freely. By default, however, the log file contains information

about the processing and morphological decomposition of the phonological words in the input, application of the ranking principles leading into Merge-1, transfer operation applied to the final structure when no more input words are analyzed and many aspects of semantic interpretation. Intermediate left branch transfer operations are not reported in detail. The beginning of a log file is illustrated in Figure 20.

```

3  #1. Pekka nukkuu
4  ['Pekka', 'nukkuu']
5
6  1. ['Pekka', 'nukkuu']
7
8      Next item: "Pekka". Lexical retrieval...(55ms) Word "Pekka-#D#sg#3p#def#hum#[{-nom}]" contains mul
9      Next item: "hum$". Lexical retrieval...(45ms) Inflectional feature hum$...(50ms) Added ['LANG:FI
10     Next item: "def$". Lexical retrieval...(45ms) Inflectional feature def$...(50ms) Added ['LANG:FI
11     Next item: "3p$". Lexical retrieval...(35ms) Inflectional feature 3p$...(40ms) Added ['LANG:FI',
12     Next item: "sg$". Lexical retrieval...(35ms) Inflectional feature sg$...(40ms) Added ['LANG:FI',
13     Next item: "D$". Lexical retrieval...(25ms) Adding inflectional features ['LANG:FI', 'NOM', 'PHI
14     Item enters active working memory. Computing semantics for D(IDX:1,QND)...Applying semantic crit
15
16     Next item: "Pekka-". Lexical retrieval...(65ms) Done.(70ms)
17     Item enters active working memory.
18
19  2. Consume "Pekka": D + Pekka
20     One solution due to sinking.(75ms) Done.
21     Results:
22     |      |      (1) D
23     Sinking Pekka into D = D(N) (80ms)
24     Next item: "nukkuu". Lexical retrieval...(65ms) Word "nukku-#T/fin#[{-V}]" contains multiple morph
25     Next item: "T/fin$". Lexical retrieval...(65ms) Adding inflectional features ['LANG:FI', 'PHI',
26     Item enters active working memory.
27
28  3. Consume "T": D(N) + T
29     Working memory operation...1 nodes currently in active memory.
30     Filtering and ranking merge sites...Filtering...Done. Ranking...Bottom-up baseline ranking...+Sp
31     Results:
32     |      |      (1) D(N)
33     Now exploring solution [D(N) + T]...Transferring left branch D(N)...(80ms) Interpreting [D Pekka
34     Result: [[D Pekka] T]...Done.
35
36     Next item: "nukku-". Lexical retrieval...(65ms) Done.(70ms)
37     Item enters active working memory.
38
39  4. Consume "nukku": [[D Pekka] T] + nukku
40     One solution due to sinking.(75ms) Done.
41     Results:
42     |      |      (1) T
43     Sinking nukku into T = [[D Pekka] T(V)] (80ms)

```

Input sentence

Morphological decomposition

Lexical item arrives to syntactic module

Merge-1

Figure 20. Screen capture from the log file.

Figure 21 illustrates transfer, LF-interface calculations and post-syntactic processes, when the input sentence is *John admires Mary*.

73	Trying spellout structure [[D John] [T(v,V) D(N)]]	Candidate solution
74	Checking surface conditions...Done.	
75	Transferring to LF...(75ms)	
76	1. Head movement reconstruction...Reconstruct v(V) from within	Head reconstruction
77	= [[D John] [T [v [admire [D Mary]]]]](90ms).	
78	2. Feature processing...Done.	
79	= [[D John] [T [v [admire [D Mary]]]]](90ms).	
80	3. Extraposition...Done.	
81	= [[D John] [T [v [admire [D Mary]]]]](90ms).	
82	4. Floater movement reconstruction...Done.	Adjunct reconstruction
83	= [[D John] [T [v [admire [D Mary]]]]](90ms).	
84	5. Phrasal movement reconstruction...(95ms) (100ms) Done.	Phrasal reconstruction
85	= [[D John]:2 [T [v [admire [D Mary]]]]](100ms).	
86	6. Agreement reconstruction...(105ms) (105ms) T acquired PHI:	Agreement reconstruction
87	= [[D John]:2 [T [v [admire [D Mary]]]]](105ms).	
88	7. Last resort extraposition...(110ms) LF-interface test...Done.	
89	= [[D John]:2 [T [v [admire [D Mary]]]]](110ms).	
90	Done.	
91	LF-legibility check...Checking LF-interface conditions...(115ms) LF-interface	LF legibility and semantic interpretation
92	Interpreting TP globally:"T" with ['PHI:DET:'] was associated at LF with 1.	
93	Narrow semantics for DP: Project (1, QND) for DP ([D John])...Project (2,	
94	Narrow semantics for vP: Project (3, PRE) for v...Project (4, GLOBAL) for	
95	Narrow semantics for admireP: Project (5, PRE) for admire...Project (6,	
96	Narrow semantics for DP: Project (7, QND) for DP ([D Mary])...Project (8,	
97	Denotations:	Possible denotations
98	[D John]~['2']	
99	[D Mary]~['8']	
100	Assignments:	Assignments
101	Assignment {'1': '2', '7': '8'} (R=set()) accepted.Calculating information	
102	Solution was accepted at 1135ms stimulus onset.	
103		
104	Semantic interpretation:	
105	Recovery: ['Agent of T(John)', 'Agent of v(John)', 'Patient of admire(Mary)']	
106	Aspect: []	
107	DIS-features: []	
108	Operator bindings: []	
109	Semantic space:	Summary of semantic interpretation
110	Speaker attitude: []	
111	Assignments:	
112	[D John] ~ 2, [D Mary] ~ 8, Weight 1	
113	Information structure: {'Marked topics': [], 'Neutral gradient': [['D John'], ['D Mary']], 'Marked topics': []}	
114	D-features: []	

Figure 21. Part of the derivational log file.

Lines 76-90 illustrate transfer and show all subprocesses that take place. They correspond to well-known linguistic processes (head reconstruction, extraposition, adjunct reconstruction, phrasal \bar{A}/A reconstruction, agreement reconstruction and last resort extraposition). The results of LF-legibility calculations are shown on line 91, which is followed by operations that take place inside narrow semantics. The numbers shown in connection with denotations (lines 98-99) refer to semantic objects in the global discourse space when processing ended, which are also listed in the derivational log file (not shown in Figure 21). These objects are projected into existence non-incrementally during post-syntactic processing (lines 93-96) for reasons explained in Section 4.10.2. In a more realistic model, these operations are performed on a phase-by-phase basis. Final assignments and their weights are also listed in the summary of semantic interpretation (see line 112). What follows in the log file after line 114 are list of all lexical items used in the calculations and their lexical features and the full contents of semantic inventories.

6.4.3 Simple logging

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

6.4.4 Saved vocabulary

Each time a study is run, the program takes a snapshot of the surface vocabulary (lexicon) as it stands after all processing has been done (after each sentence has been processed) and saves it into a separate text file with the suffix `_saved_vocabulary.txt`. The reason is because the ultimate lexicon used in each study is synthesized from three sources (language-specific lexicon, universal morphemes and lexical redundancy rules) and thus involves computations and assumptions whose output the user might want to verify. Notice that the complete feature content of each terminal element that occurs in any output solution is stored into the log file together with the solution (Section 6.4.2) and does not appear in this file.

6.4.5 Images of the phrase structure trees

The algorithm stores the parsing output in phrase structure images (PNG format) if the user activates the corresponding functionality. The function can be activated by input parameters in the study configuration file (Section 6.3.4). Figure 23 illustrates the phrase structure representation generated for a simple transitive clause in English, when produced without any lexical information.

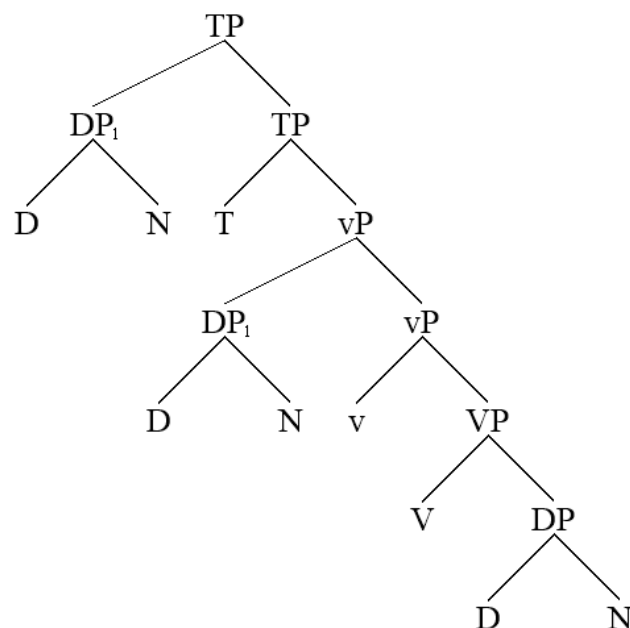


Figure 23. A simple phrase structure image generated by the algorithm for a simple transitive clause *John admires Mary*.

The lexical category labels shown in these images are drawn from the list of major categories defined at the beginning of the `phrase_structure.py` module. If the label of an element is not recognized, it will appear as X (and XP for phrases).

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

6.4.6 Resources file

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

Resource consumption is reported in two places. A summary is normally provided in the results file. In addition, the algorithm generates a file with the suffix “_resources” to the study folder that reports the results in a format that can be opened and processed directly with external programs, such as MS Excel or by using external Python libraries such as pandas or NumPy. The list of resources reported is provided in the parser class and can be modified there. In addition to listing resource consumption, each line contains also the study number (as specified in the input parameters) and the numerical classifications read from the test corpus file, if any (see Section 6.3.1).

6.4.7 Semantic interpretation

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

6.5 Main script

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

```
# Prepare file systems and logging
local_file_system.initialize(args)
local_file_system.configure_logging()
```

The first line initializes the whole simulation by using the parameters provided by the user. All output files are prepared here for writing. The second line prepares logging. Next, the script prepares parsers for all input languages.

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

It checks what languages are present in the lexicon and then prepares a “brain model” for each language (currently Finnish, English and Italian). This makes the processing more efficient, so that we don’t need to reconstruct the parser each time a new language is presented in the input. A brain model is an instantiation of the linear phase model (the parser) together with language specific parameters, thus notice that it takes language as an input parameter, which then becomes a “contextual parameter” of the model. It is assumed that a bilingual speaker can change this contextual parameter depending on the language being used; at the level of code, we just change the brain model (e.g., `parser_for[Finnish]` → `parser_for[English]`) on the basis of the language in the input sentence. The computational core of the model has no language-specific parameters; rather, they came into effect when the functional lexicon is processed through lexical redundancy rules. The issue is empirically nontrivial, since the question of how language comprehension systems differs from language to language is controversial. Currently, it is assumed that only the functional lexicon changes. When the lexicon is loaded (`load_lexicon_()`), all lexical items that

do not have specification for language – usually these are only functional items – will be provided one by using the contextual language parameter:

```
if not {f for f in lexical_features if f[:4] == 'LANG'}:
    lexical_features.append(self.language)
```

This will trigger language-specific redundancy rules when these rules are applied.

```
new_const.features = self.apply_parameters(self.apply_redundancy_rules(lexical_features))
```

Apply_parameters() function is a residuum from an older model and only contains few parametric changes that are mostly irrelevant. The function remains in the model, however, because I am not certain that it will not be needed. Once the model is loaded, it will be initialized, which resets all its internal data structures.

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

```
sentences_to_parse = [(sentence, group, part_of_conversation)
                      for (sentence, group, part_of_conversation)
                      in local_file_system.read_test_corpus()]
```

The test corpus reader returns tuples that contain the sentence, its experimental group number and whether it is part of a conversation. The user can add any information required to the reader (e.g., contextual tags) and then change the code here to correspond to these changes. These sentences are then forwarded to the parser brain model, one at a time.

```
sentence_number = 1
for sentence, experimental_group, part_of_conversation in sentences_to_parse:
    if not is_comment(sentence):
        language = lang_guesser.guess_language(sentence)
        local_file_system.print_sentence_to_console(sentence_number, sentence)
        parser_for[language].parse(sentence_number, sentence)
        local_file_system.save_output(parser_for[language],
                                     sentence_number,
                                     sentence,
                                     experimental_group,
                                     part_of_conversation)
        if not part_of_conversation:
            parser_for[language].narrow_semantics.global_cognition.end_conversation()
        sentence_number = sentence_number + 1
    else:
        local_file_system.parse_and_analyze_comment(sentence)
        local_file_system.write_comment_line(sentence)
```

For each sentence this function tries to guess the language, prints the sentence to console, sends it for the parser and saves the output when the parser has finished the job.

7 Grammar formalization

7.1 Basic grammatical notions (phrase_structure.py)

7.1.1 Introduction

The class `PhraseStructure` (defined in `phrase_structure.py`) defines the phrase structure objects called constituents that are manipulated at each stage of the processing pipeline. The organization of the functions and their use is determined both by the theory and implementation and code comprehension. Some functions that appear here are called only once from some other module, yet they are defined in this class for readability; which class will ultimately define them, if any, is a matter of empirical research.

7.1.2 Types of phrase structure constituents

7.1.2.1 Lexical items

Lexical items are sets of lexical features. Once they enter syntax, they are associated with a phrase structure node (constituent). Thus we can think of the lexical component as providing feature sets that are attached to syntactic nodes in the syntactic component, like fish that get caught in a net. Phrase structure nodes define the phrase structure geometry. Notice that the feature sets themselves are not constituents.

7.1.2.2 Primitive and terminal constituents

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

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

It follows that a constituent that has zero or one immediate daughter constituents is primitive. A constituent is complex if it is not primitive (next section). Thus we have a three-way classification: phrasal constituents versus nonphrasal constituents, where the latter is further divided into terminal and nonterminal constituents. Nonterminal nonphrasal constituent correspond and are used to represent complex heads (Section 4.8.5). All nonphrasal constituents can be associated with lexical feature sets.

The decision to impose a rigid left-right constituent structure – that is, explicit left and right constituents – is not arbitrary. It is assumed that the phrase structure is binary branching, and the assumption that each complex constituent can have the left and right daughter constituents but no more captures this axiom. If we used some more productive data structures, such as lists or sets, then we would need an additional axiom restricting the number into two, which does not seem right to me.

7.1.2.3 Complex constituents; left and right daughters

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

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

Phrasal constituents are in the domain of phrasal syntactic rules. As stated, a complex constituent can have a *left constituent* and a *right constituent*. These notions are defined as follows.

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

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

The reason we check `self.mother` is because the second condition would raise an error if it were applied to the root node.

7.1.2.4 Complex heads and affixes

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

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

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

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

Complex heads and their existence in the model is nontrivial both theoretically and empirically. The way they are represented here is a compromise: we need them in order to calculate certain nontrivial datasets, but we do not want to expand the phrase structure geometry with additional novel devices or representations. Hence a defective phrase structure geometry was used. The algorithm notates the output as $X(Y)$, meaning that morpheme Y is inside X and hence $[_X Y]$. We

could use some other implementation if there were empirical or theoretical reason to do so while keeping the neutral notation. See Section 4.8.5.

7.1.2.5 Sisters

Sisterhood relation is not primitive and is defined by relying on the mother-of relation. Two constituents are *geometrical sisters* if they occur inside the same constituent, i.e. have the same mother. The right constituent constitutes the geometrical sister of the left constituent, and vice versa.

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

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

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

This definition ignores invisible (adjoined) right constituents. The fact that sisterhood relation is defined in this way plays a role everywhere. See Section 4.6 for nontechnical description of these issues.

7.1.2.6 Complement, proper complement

Complement is a local grammatical dependency that is used mainly for selection purposes. A *proper complement* of a constituent is its right sister if one exists.

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

This function applies only to nonphrasal elements, because the notion of complement is applied only to lexical elements that can have complement selection features. A (regular) *complement* is defined as the same relation as sisterhood. Proper complements and complements are used for slightly different purposes. Geometrical sisterhood plays no role in defining complements, because it does not distinguish between adjuncts and other constituents.

7.1.2.7 Labels

The labeling algorithm is (65).

(65) Labeling

Suppose α is a complex phrase. Then

- a. if the left constituent of α is primitive, it will be the label; otherwise,
- b. if the right constituent of α is primitive, it will be the label; otherwise,
- c. if the right constituent is not an adjunct, apply (15) recursively to it; otherwise,
- d. apply (15) to the left constituent (whether adjunct or not).

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

7.1.2.8 Minimal search, geometrical minimal search and upstream search

Several operations require that the phrase structure is explored in a pre-determined order. The *minimal search* from α explores the right edge of α in a downstream direction: left branches are phases and are not visited. Minimal search on α constitutes an operation that crawls downwards on the right edge of α . It is defined by creating an iteration over phrase structure. Thus, the definition for minimal search itself is given as follows.

```
def minimal_search(self):
    return [node for node in self]
```

The more important part is contained in `__getitem__` that enumerates the constituents of a phrase structure. The name `__getitem__` is based on Python naming conventions and cannot be changed.

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


The logic of this function as follows. It orders certain nodes in the phrase structure so that we can count them, starting from 0 which is the root/starting point and ending with the bottom right node. This counting is not part of the theory but concerns implementation, so that we can establish identities between nodes in two distinct phrase structure objects. We can imagine every step in the minimal search to be defined by a unique number that tells how the steps are ordered. To do the numbering, we descend the phrase structure by using the four principles listed in the comments. First we try to follow labelling through the right constituent if possible, taking care of $[_{YP} XP YP]$ and $[_{YP} XP Y]$. In either case, the right constituent projects. If neither is true, then the left constituent must project. If it is complex, then we descend into it, handling $[_{XP} XP \langle \alpha \rangle]$ where $\langle \alpha \rangle$ is an adjunct. If this does not work, the left constituent must be primitive. If the right constituent is an adjunct, we select the primitive lexical item that will terminate the search; otherwise we select the right constituent, primitive or complex, that is selected by X. This handles $[_{XP} X \langle \alpha \rangle]$ and $[_{XP} X YP]$. The interpretation is that once we can no longer search by looking for the head we follow selection.

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

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

7.1.3 Upward paths

The model makes much use of various upward paths that we would ideally like to unify. A promising unification is Kayne's (Kayne 1983, 1984) path approach. Suppose that α constitutes a probe that triggers the scanning operation. First we define the notion of *working memory path*.

```
def working_memory_path(probe):
    node = probe.mother
    working_memory = []
    #=====
    while node:
        if node.left_const.head() != probe:
            working_memory.append(node.left_const)
            node = node.walk_upwards()
    #=====
    return working_memory
```

Starting from the probe α , we proceed upwards and collect all left constituents into a list, ignoring the probe itself. Upward walking follows dominance relations by ignores right adjuncts, which are pulled into separate working space and are invisible.

```
def walk_upwards(self):
    node = self.mother
    if self.is_left():
        while node and node.right_const.adjunct:
```

```

        node = node.mother
    return node

```

These functions implement a “memory scanning” operation. *Edge* refers to a smaller and more local segment of the path, and contains all elements in the path that are inside the projection from α .

```

def edge(probe):
    return list(takewhile(lambda x:x.mother.head() == probe, probe.working_memory_path()))

```

Here we take all elements β from the working memory path as long as the label of the mother of β is α . Thus, if XP in $[_{YP} Y [_{\alpha P} XP \alpha P]]$ is inside the path from α , it will be the last element inside the edge of α . The operation collects all specifiers from αP .

7.1.4 Basic structure building

7.1.4.1 Cyclic Merge

Simple Merge takes two constituents α , β and yields $[\alpha, \beta]$, α being the left constituent, β the right constituent. It is implemented by the class constructor `__init__()`, which takes α and β as arguments and return a new constituent. It sets up the mother-of relations for both sisters.

```

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

```

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

Notice that all constituents, whether primitive or not, can have a set of features. This set is currently used only for lexical items, but was originally generalized so that it still applies to all constituents. Crucially, a complex head $[\alpha \beta]$ now takes advantage of this option: here $[\alpha \beta]$ is associated with the lexical features of α , and we interpret the constituency relation as creating a

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

7.1.4.2 Countercyclic Merge-1

Countercyclic Merge-1 (`merge_1(α , β , direction)`) targets constituent α inside an existing partial phrase structure and creates a new constituent γ by merging β either to the left or right of α : $\gamma = [\alpha, \beta]$ or $[\beta, \alpha]$. Thus, if we have a phrase structure $[X... \alpha ... Y]$, then Merge-1 generates either (a) or (b).

(66)

- a. $[X...[_\gamma \alpha \beta]...Y]$
- b. $[X...[_\gamma \beta \alpha]...Y]$

Constituent γ then replaces α in the phrase structure, with the phrase structural relations updated accordingly. Both Merge to the right and left, and both countercyclically and by extending the structure, are allowed. The range of options is compensated by the restricted conditions under which each operation can occur. The fact that Merge-1 dissolves into separate processes is reflected in the code, which contains three separate functions: the first (`local_structure()`) obtains a snapshot of the local structure around α (its mother and position in the left-right axis), the second creates $[\alpha \beta]$ or $[\beta \alpha]$ (`asymmetric_merge()`), and the third (`substitute()`) substitutes α with the new constituent $[\alpha \beta]$ by using local constituent relations recorded by the first operation.

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

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

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

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

7.1.4.3 Remove

An inverse of countercyclic Merge-1 is remove (α .remove()), which removes constituent α from the phrase structure and repairs the hole. This operation is used when the parser attempts to reconstruct movement: it merges elements into candidate positions and removes them if they do not satisfy a given set of criteria.

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

7.1.4.4 Detachment

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

```
def detach(self):
    is_right = self.is_right()
    original_mother = self.mother
    self.mother = None
    return original_mother, is_right
```

7.1.5 Nonlocal grammatical dependencies

7.1.5.1 Probe-goal: probe(label, goal_feature)

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

(67) Probe-goal

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

For everything else than criterial features, G must be found from a primitive head to the left of the right edge node. The present implementation has an intervention clause which blocks further search if a primitive constituent is encountered at left that has the same label as the probe, but the matter must be explored in a detailed study.

```
def probe(self, feature, G):
    def inside_path(node):
        if node.is_primitive():
            return node
        return node.left_const.head()

    if self.sister():
        # ----- minimal search -----
        for node in self.sister():
```

```

    if G in inside_path(node).features:
        return True
    if G[:4] == 'TAIL' and G[5:] in node.left_const.scan_criterial_features():
        return True
    if feature.issubset(inside_path(node).features):
        break
# -----

```

The operation searches for feature G from the vicinity of each node (function `inside_path(node)`) that is within the minimal search path. There is an intervention condition which terminates search (break) if an intervention feature is encountered. The intervention feature (feature) can be set at the function call.

The probe-goal mechanism is used to implement nonlocal selection. One example of nonlocal selection is the relationship between D and N. Every D must be paired with an N, but the relationship can be intervened by a number of other functional heads such as Q, Num and others. It could be possible to implement the grammar without the probe-goal function, but so far I have been unable to eliminate it.

7.1.5.2 Tail-head relations

A tail-head dependency is formed when a probe P must locate a goal G either inside the head of its own projection (“strong tail test”) or inside an upward path by memory scanning (22)(“weak tail test”). Whether both or only one test must be performed can be defined by a lexical feature, examined below. 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* $F = \{f_1 \dots f_n\}$ which must all be checked in order for the dependency to form. The main function is as follows.

```

def tail_test(self):
    positive_features = {f for f in self.get_tail_sets() if self.positive_features(f)}
    negative_features = {f for f in self.get_tail_sets() if self.negative_features(f)}
    checked_positive_features = {tail_set for tail_set in positive_features if
        self.strong_tail_condition(tail_set) or self.weak_tail_condition(tail_set)}
    checked_negative_features = {tail_set for tail_set in negative_features if
        self.strong_tail_condition(tail_set) or self.weak_tail_condition(tail_set)}
    self.consume_resources('Tail test')
    return positive_features == checked_positive_features and not checked_negative_features

```

The function collects positive and negative target features (tail sets, i.e. F above) into their own groups, and then performs the weak and strong tests and collects the checked positive features and checked negative features against into their own results groups. The test passes if all positive features were checked and none of the negative features were. To illustrate, suppose the positive tail test targets features {A, B} and the negative tail test targets features {C, D}. Then any head with both features {A, B} is accepted, any head with neither features or only one of them will be rejected. Furthermore, any head with either feature C, D or both will cause rejection. This means that a head with {A, B, C} will also cause rejection: the positive test passes but the negative does not.

The code for the strong tail test is as follows:

```
def strong_tail_condition(self, tail_set):
    if '$NO_S' not in tail_set and self.max() and self.max().mother:
        return self.max().container().match_features(tail_set).outcome or \
            (self.max().mother.sister() and
             self.max().mother.sister().match_features(tail_set).outcome)
```

The first test controls for the feature [\$NO_S] (“no strong test”) which, if it is part of the tail set, will block the strong test. If the test proceeds, we examine if the head α of the containing projection αP has the required features. The second clause is a stipulation that is required to handle certain edge cases. The weak test as follows:

```
def weak_tail_condition(self, tail_set):
    if '$NO_W' not in tail_set:
        if {'φ', 'D'} & self.features:
            self.consume_resources('Case checking')
            for m in (affix for node in self.working_memory_path() if node.is_primitive() for affix in
                      node.get_affix_list()):
                test = m.match_features(tail_set)
                if test.match_occurred:
                    return test.outcome
    if self.negative_features(tail_set):      # Unchecked negative features will pass the test
        return False
```

Feature [\$NO_W] (“no weak test”) serves again as the gate feature and blocks the operation if needed. We then examine all morphemes (affixes) inside all constituents in the upward path and check whether the features match, and if they do, the result is positive. If a negative feature is matched, the result is negative. Both strong and weak tests use the feature match function:

```
def match_features(self, features_to_check):
    if self.negative_features(features_to_check) & self.features:
        return Result(True, True)
        # Match occurred, outcome positive
    if self.positive_features(features_to_check) & self.features:
        return Result(True, self.positive_features(features_to_check).issubset(self.features))
        # Match occurred, outcome negative (partial match)/positive (full match)
    return Result(False, None)
    # No match occurred, no outcome (usually evaluates into False)
```

The result is expressed as a tuple where the first element expresses whether a match occurred, and the second what the outcome was (success, fail). Notice that a match can fail if only a subset of the features were matched, or if the feature is negative.

The shape of these functions is determined by two factors: empirical data and simplicity. The reason the two test types, strong and weak, exists is because while most dependencies rely on the weak test (case checking, control, binding), adverb placement, and perhaps also some case checking operations, requires the stronger test. It is therefore possible that two separate mechanisms are at stake, though it still seems that they share many features, such as feature matching. The existence of negative features are likewise suggested by empirical data, such as the behavior of negative and positive polarity items, but here there is an alternative: instead of checking the absence of some feature, we could check the presence of the mirror feature –

provided that such feature is always present. In other words, the cost of removing negative feature checking from this function is that we need to stipulate the positive feature everywhere.

7.2 Transfer (transfer.py)

7.2.1 Introduction

Reconstruction is a reflex-like operation that is applied to a phrase structure and takes place without interruption from the beginning to the end. From an external point of view, it constitutes ‘one’ step; internally the operation consists of a sequence of steps:

```
def transfer(self, ps, embedding=3):
    output_to_interfaces['spellout structure'] = ps.copy()

    ps = self.head_movement_module.reconstruct(ps)
    output_to_interfaces['surface structure'] = ps.copy()

    self.feature_process_module.disambiguate(ps)
    self.extraposition_module.reconstruct(ps)
    ps = self.floaters_movement_module.reconstruct(ps)
    output_to_interfaces['s-structure'] = ps.copy()

    self.phrasal_movement_module.reconstruct(ps)
    self.agreement_module.reconstruct(ps)
    self.extraposition_module.last_resort_reconstruct(ps)
    output_to_interfaces['LF structure'] = ps.copy()

    return ps, output_to_interfaces
```

As this sequence shows, the model maps spellout structures arriving from the parser into surface structures (by head reconstruction) → s-structures (by floaters reconstruction and few other operations) → LF-structures (by A/ \bar{A} reconstruction). In some earlier models, there was a single interface between syntax and semantics at the endpoint of the syntactic pathway (LF-structure), but this proved to be too simple, hence this sequence sends information to the semantic module via several interfaces through the data structure `output_to_interfaces`. These structures are visible in the results and derivational log files per each input sentence.

7.2.2 Head reconstruction (head_reconstruction.py)

Head reconstruction is the first operation performed during transfer. All other components of transfer presuppose that all grammatical heads are at their canonical positions. In addition, morphological chunking has many properties which suggest that the operation takes place relatively close to the surface lexicon and sensorimotoric interface(s).

First we separate two situations: one in which (i) the reconstruction targets a complex head in isolation, and another in which it (ii) targets a complex phrase that may or may not contain complex heads. The former (i) occurs when the parser considers some solution $[\alpha(\beta) + \phi]$ and we must transfer $\alpha(\beta)$ in order to evaluate whether $[\alpha(\beta) + \phi]$ is acceptable and/or likely. Transfer will then target $\alpha(\beta)$, but the problem is that this situation involves branching between two possible solutions $[[\alpha \ \beta] \ \phi]$ or $[\alpha \ [\beta \ \phi]]$. In earlier models head reconstruction itself contained a

limited degree of ‘parsing intelligence’ that determined the choice; the current version includes both options into the parsing search tree. These options are visible in the derivational log file. When the parser consider a merge operation $\alpha + \beta$, notation $\alpha\downarrow + \beta$ means that α will be transferred before merge. This is the default option, and is applied always when α is a phrase. If the arrow is not present, α will not be transferred. This solution is currently only considered in connection with complex heads. The result will be that the complex head will get reconstructed later when the hosting phrase gets transferred.

Standard reconstruction is performed inside function `reconstruct()`. The algorithm targets the bottom node and works its way upwards until there is no more structure. For each dominating node it encounters on its way up, it detects complex heads and attempts to reconstruct them.

```
def reconstruct(self, phrase_structure):
    current_node = phrase_structure.bottom()
    while current_node.mother or self.detect_complex_head(current_node):
        current_node = self.consider_head_reconstruction(current_node,
            self.detect_complex_head(current_node))
    return phrase_structure.top()
```

Head reconstruction is implemented as follows. First we consider whether it applies, given some node:

```
def consider_head_reconstruction(self, current_node, targeted_head):
    if targeted_head:
        current_node = self.create_head_chain(targeted_head,
            self.determine_intervention_features(targeted_head), self.get_affix_out(targeted_head))
        return current_node
    return current_node.mother
```

If it does, then a head chain creation algorithm is called with three arguments: the complex head $X(Y)$ itself, intervention feature, and the affix Y that needs reconstruction. Head chain is created as follows:

```
def create_head_chain(self, complex_head, intervention_feature_set, affix):
    if not complex_head.sister() or complex_head.is_right():
        complex_head.merge_l(affix, 'right')
        return affix

    # ----- minimal search -----#
    starting_pos_node = complex_head.sister()
    for node in starting_pos_node:
        if node != starting_pos_node and intervention_feature_set & node.sister().features:
            node = starting_pos_node # Reset the search pointer after intervention
            break
        node.merge_l(affix, 'left')
        if self.reconstruction_is_successful(affix):
            self.brain_model.consume_resources("Move Head")
            return affix
        affix.remove()
    # -----#

    if not self.consider_right_merge(affix, node, starting_pos_node):
        starting_pos_node.merge_l(affix, 'left') # last resort
        self.brain_model.consume_resources("Move Head")

    return affix
```


The first part takes care of an edge case where we are targeting an isolated head $X(Y)$ without no surrounding structure. The result will be $[X(__1) Y_1]$. This should of course be unified with the general rule. If there is structure to the right of the targeted complex head $[X(Y) Z]$, we select it (Z) and use minimal search to try out all possible solutions. There are three exit conditions: (i) intervention occurs, in which case search is terminated; (ii) a legitimate position is found; (iii) we reach the end of the structure without solution. If a legitimate position is found, option (ii), the head will remain at that position, a head chain was created, and the execution of the function ends; conditions (i, iii) are failures in the sense that a solution was not found but the search ended. This, then, leads into the last resort options written at the end of the function. One of these solutions is always adopted, which means that all targeted complex heads will always get reconstructed.

Reconstruction succeeds if and only if the head can be selected at that position and one extra condition is satisfied. They are stated in the following function:

```
def reconstruction_is_successful(self, reconstructed_affix):
    return reconstructed_affix.features &
        reconstructed_affix.selector().bottom_affix().licensed_complements() and \
        not self.extra_condition_violation(reconstructed_affix)
```

It follows that every head will be reconstructed to the first position where it satisfies the two conditions independent of whether this solution will be judged legitimate at some later step. This implements many locality restrictions on head reconstruction (and head movement) visible in the data. The first condition requiring selection is trivial: we can reconstruct Y into a gap position where a higher head selects it. The second condition is nontrivial and stated as follows:

```
def extra_condition_violation(self, affix):
    return 'C/fin' in affix.selector().features and affix.EPP() and not affix.edge()
```

It applies for the special case of $C(T_{EPP})$ and takes care of a situation where the finite EPP-head T is reconstructed so that a grammatical subject, if present, occurs between it and a governing finite C . It allows finite T to descend into a lower position to satisfy its EPP feature. Without this rule, T would be reconstructed to $[C [T\dots]]$, leaving the possible grammatical subject into a postverbal position. If there is no grammatical subject, the last resort rule will still generate the correct solution $[C [T\dots]]$. The reason this rule is stated as a special condition is because I was unable to calculate the whole dataset(s) if the condition was removed or generalized. A third option would be to posit a special reconstruction rule that brings the subject to the EPP-position.

Head chain creation requires that we specify the intervention feature. There are two separate cases. If the target head contains an operator feature and constitutes a concept in the semantic sense, then ϕ and all operator features cause intervention. In all other cases, intervention is caused by all functional heads.

```
def determine_intervention_features(self, head):
    if self.brain_model.narrow_semantics.is_concept(head) and {feature for feature in head.features
    if feature[:2] == 'OP'}:
        return {'φ'} |
        set(self.brain_model.narrow_semantics.operator_variable_module.operator_interpretation.keys())
    return {'!COMP:*'}
```

Functional heads are defined as such by the feature [!COMP:*] which states the requirement that they must have a complement, a kind of inverse EPP condition. The second condition captures Travis’ Head Movement Constraint (HMC). The first condition simulates long head movement, in Finnish in particular.

7.2.3 Adjunct reconstruction (*adjunct_reconstruction.py*)

Adjunct reconstruction performs minimal search from the top of the structure, detects adjuncts and reconstructs them. The code that handles these operations is below.

```
def reconstruct(self, ps):
    for node in [node for node in ps.top()]:
        f = self.get_floater(node)
        if f:
            self.drop_floater(f)
            if f.is_right():
                break
    return ps.top()
```

The first function returns an adjunct to be reconstructed, while the second reconstructs (“drops”) it.¹⁴ Floater detection is performed for both the left A and right B constituent of any node [A B] visited by the minimal search.

```
def get_floater(self, ps):
    if self.detect_floater(ps.left_const):
        [. . .]

    if self.detect_floater(ps.right_const):
        [. . .]
```

where floater is a complex phrase structure that has not been copied elsewhere, has tail features, is adjoinable and can float (reconstruct):

```
def detect_floater(self, ps):
    return ps and \
        ps.is_complex() and \
        not ps.find_me_elsewhere and \
        ps.head().get_tail_sets() and \
        'adjoinable' in ps.head().features and \
        '-adjoinable' not in ps.head().features and \
        '-float' not in ps.head().features and \
        not
self.controlling_parser_process.narrow_semantics.operator_variable_module.scan_criterial_features(ps)
```

¹⁴ The reason we must exit the function if the node was right is because we want to avoid the situation where the minimal search descends inside an adjunct. Suppose we are targeting node $\alpha = [A B]$ and find that B is an adjunct that must be reconstructed. In this situation we do not want to descend inside B.

The property ‘±adjoinable’ licenses the adjunction operation, whereas ‘±float’ licenses reconstruction/floating. Once a potential floater is detected, we must find out if it requires reconstruction. For left floaters, reconstruction is attempted if and only if tail-test fails or the adjunct occurs in a finite EPP position:

```
if self.detect_floater(ps.left_const):
    H = ps.left_const.head()
    if not H.tail_test():
        log(ps.left_const.illustrate() + ' failed ' + illu(H.get_tail_sets()) + '. ')
        return ps.left_const
    if ps.left_const.container():
        J = ps.left_const.container()
        if (J.EPP() and 'FIN' in J.features) or ('-SPEC:' in J.features and ps.left_const ==
next((const for const in J.edge()), None)):
            return ps.left_const
```

For the right adjuncts, reconstruction is attempted if the tail-test fails. Floater reconstruction is defined as follows:

```
def drop_floater(self, original_floater):
    [...]
    test_item = original_floater.copy()
    local_tense_edge = self.local_tense_edge(original_floater)
    # ----- minimal search -----#
    for node in local_tense_edge:
        if termination_condition(node, original_floater, local_tense_edge):
            break
        self.merge_floater(node, test_item)
        self.adjunct_constructor.externalize_structure(test_item)
        if self.validate_position(test_item, starting_point_head):
            test_item.remove()
            dropped_floater = self.copy_and_insert_floater(node, original_floater)
            self.controlling_parser_process.narrow_semantics.pragmatic_pathway.
            unexpected_order_occurred(dropped_floater, starting_point_head)
            return
        test_item.remove()
    # -----#
```

First we locate the local minimal tense boundary, which accounts for the fact that adjuncts cannot float out of their own finite clauses. Local tense boundary is searched by scanning the working memory (upward path):

```
def local_tense_edge(self, ps):
    return next((node.mother for node in ps.working_memory_path() if {'T/fin', 'FORCE'} &
node.features), ps.top())
```

Then we perform minimal search from that node and locate the first valid position, notify the pragmatical pathway of the discovery and (re)merge the adjunct at that position. A valid position is defined differently for left and right adjuncts:

```
def validate_position(self, test_item, starting_point_head):
    return self.conditions_for_right_adjuncts(test_item) or \
        self.conditions_for_left_adjuncts(test_item, starting_point_head)
```

For right adjuncts, we only need to check if the tail test succeeds. For left adjuncts, there is a sequence of additional (mostly stipulated) tests that we will ideally want to reduce away:

```
def conditions_for_left_adjuncts(self, test_item, starting_point_head):
    if test_item.head().tail_test():
        if not test_item.container():
            return True
```

```

        if 'GEN' in test_item.head().features and 'φ' not in test_item.container().features:
            return True
        if test_item.container() == starting_point_head:
            return False
        if '-SPEC:*' in test_item.container().features:
            return False
        if 'φ' in test_item.head().features and not
self.controlling_parser_process.LF.projection_principle(test_item.head(), 'weak'):
            return False
        return True

```

These tests rule out certain edge cases, for example, a situation where an otherwise valid position constitutes a specifier position for a head that cannot carry any specifiers (fourth condition) or is not possible by the projection principle when the reconstructed phrase is a referential argument (fifth condition). In addition, we do not want to reconstruct that adjunct inside the same projection where it started (third condition).

Notice that what ultimately controls the distribution of adjuncts are the tail-features. Thus it is expected that the semantic component contains a corresponding mechanism that uses the tail-features for interpretation. It has not been implemented.¹⁵ Lexical features \pm adjoinable and \pm float are used when we need to dissociate tail-features from adjoinability and floating. English accusative pronouns, for example, must be tail-checked but they do not adjoin or float.

7.2.4 \bar{A} -reconstruction (*phrasal_reconstruction.py*)

\bar{A} -reconstruction detects oddball phrases with operator features and reconstructs them downstream by minimal search. The minimal search travels through the structure, pull oddball phrases into memory buffer from nonthematic EPP positions while it searches suitable positions for elements that are already stored in the said buffer. These operations take place simultaneously, so that there is only one minimal search per transfer.

```

def reconstruct(self, ps):
    self.brain_model.syntactic_working_memory = []
    # ----- minimal search -----#
    for node in ps:
        if self.get_local_head(node) and self.get_local_head(node).EPP():
            self.pull_into_working_memory(self.get_local_head(node))
        if self.get_local_head(node):
            self.brain_model.LF.try_LFmerge(self.get_local_head(node))
        if self.intervention(node):
            break
    # -----#

```

The operation that collects specifiers of nonthematic heads is as follows.

```

def pull_into_working_memory(self, head):
    for i, spec in enumerate(head.edge()):
        if not spec.find_me_elsewhere:
            if self.Abar_movable(spec):
                self.brain_model.syntactic_working_memory = self.brain_model.syntactic_working_memory
                    + [spec]
            else:

```

¹⁵ It is interesting to speculate what this mechanism will do for example in the case of structural case forms.

```

        self.A.reconstruct(spec)
    self.process_criterial_features(i, spec, head)

```

The function lists all phrases from the edge of the EPP head, relying on the path and working memory mechanism, and then reacts to each such phrase depending on whether it requires \bar{A} -reconstruction, thus contains an operator feature, or A-reconstruction, if not. If the former, the phrase is copied into the temporary memory buffer; if the latter, it is reconstructed immediately (see Section 7.2.5). Once a decision has been made, the function calls another function which processes the criterial (operator) features. This is where criterial features are either copied to a local head or a supporting head is generated to the structure.

```

def process_criterial_features(self, i, spec, head):
    [...]
    if i == 0:
        if not spec.find_me_elsewhere and
self.brain_model.narrow_semantics.operator_variable_module.scan_criterial_features(spec):
            head.features |= self.get_features_for_criterial_head(spec)
            [...]
    else:
        if self.specifier_phrase_must_have_supporting_head(spec):
            new_h = self.engineer_head_from_specifier(head, spec)
            spec.sister().merge_l(new_h, 'left')
            [...]

```

The first condition $i == 0$ refers to a situation where the nonthematic EPP head has only one phrasal specifier at its edge. In that case, the criterial feature is just copied to the EPP head (68)a. If there were several phrasal specifiers, then it could be that a supporting head H must be generated between them (b).

- (68) a. [YP [XP ... F ...] [Y_F ...]]
 b. [YP [XP ... F ...] H_F [ZP [Y ...]]]

Adjuncts do not need supporting heads, which is why whether a supporting head is required is defined by a separate function. The new head is generated by the function `engineer_head_from_specifier()`:

```

def engineer_head_from_specifier(self, head, spec):
    new_h = self.lexical_access.PhraseStructure()
    new_h.features |= self.get_features_for_criterial_head(spec)
    return new_h

```

The features of the new head are generated by the function `get_features_for_criterial_head()`, which will include the criterial feature itself plus several other features, such as finiteness, all processed through lexical redundancy rules.

```

def get_features_for_criterial_head(self, spec):
    criterial_features =
self.brain_model.narrow_semantics.operator_variable_module.scan_criterial_features(spec)
    if criterial_features:
        feature_set = criterial_features
        feature_set |= {'FIN', 'OP:', 'C', 'PF:C'}
        return
    self.lexical_access.apply_parameters(self.lexical_access.apply_redundancy_rules(feature_set))
    else:

```

```
return {'?'}
```

7.2.5 A-reconstruction

A-reconstruction implements local spec-to-spec movements. It uses minimal search to find the first available position for A-reconstruction and copies the phrase there.

```
def reconstruct(self, spec):
    [...]
    # -----minimal search-----#
    for node in [node for node in spec.sister()][1:]:
        if self.target_location_for_A_reconstruction(node):
            node.merge_l(spec.copy_from_memory_buffer(self.brain_model.baptize()), 'left')
            break
        if self.intervention(node):
            break
    # -----#
```

Position X is a possible target location for A-reconstruction if and only if it constitutes the sole specifier position of head H:

```
def target_location_for_A_reconstruction(self, node):
    return (node.left_const and node.left_const.is_primitive() and node.sister().is_primitive()) or
    node.is_primitive()
```

We are looking at configuration [Y _N X ...] where N = node that is surrounded by two primitive heads Y and X, and map this into [Y WP _N X ...] where WP is the reconstructed phrase. Thus, A-reconstruction cannot stack specifiers, not even if they are adjuncts. However, this function allows nonlocal A-reconstruction as long as the intervention condition is not violated and no local position exists. Minimal search is preceded by condition which specifies which phrases are candidates for A-reconstruction. In current implementation this includes licensed specifiers and VP-elements in Finnish, in order to isolate the issue of Finnish VP-fronting from the system that I deem as controversial. Licensed specifier is defined as

```
def licensed_phrasal_specifier(self):
    return next((spec for spec in self.edge()
        if 'φ' in spec.head().features and not spec.adjunct),
        next((spec for spec in self.edge()
            if 'φ' in spec.head().features and not spec.find_me_elsewhere), None))
```

which returns the most local nonadjunct φD from the edge, and if none is found, then considers the most local φD that is an adjunct; otherwise nothing. The path mechanism is again used.

7.2.6 Extraposition as a last resort (extraposition.py)

Extraposition is a last resort operation that will be applied to a left branch α if and only if after reconstruction all movement α still does not pass LF-legibility. The operation checks if the structure α could be saved by assuming that its right-most/bottom constituent is an adjunct. This possibility is based on ambiguity: a head and a phrase ‘k + hp’ in the input string could correspond to [K HP] or [K ⟨HP⟩]. Extraposition will be tried if and only if (i) the whole phrase structure (that was reconstructed) does not pass LF-legibility test and (ii) the structure contains either finiteness

feature or is a DP. Condition (i) is trivial, but (ii) restricts the operation into certain contexts and is nontrivial and possible must be revised when this operation is examined more closely. A fully general solution that applied this strategy to any left branch ran into problems.

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

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

If both tests are passed, then the operation finds the bottom HP = [H XP] such that (i) HP is adjoinable in principle and either (i.a) there is a head K such that [K HP] and K does not select HP or K obligatorily selects something else (thus, HP *should* be interpreted as an adjunct) or (i.b) there is a phrase KP such as [KP HP].¹⁶

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

HP is targeted for possible extraposition operation and HP will be promoted into adjunct (Section 7.2.7). This will transform [K HP] or [KP HP] into [K <HP>] or [KP <HP>], respectively. Only the most bottom constituent that satisfies these conditions will be promoted; if this does not work, and α is still broken, the model assumes that α cannot be fixed.

7.2.7 Adjunct promotion (*adjunct_constructor.py*)

Adjunct promotion is an operation that changes the status of a phrase structure from non-adjunct into an adjunct, thus it transfers the constituent from the “primary working space” into the “secondary working space.” Decisions concerning what will be an adjunct and non-adjunct cannot be made in tandem with consuming words from the input. The operation is part of transfer. If the phrase targeted by the operation is complex, the operation is trivial: the whole phrase structure is

¹⁶ The notion “is adjoinable” (*is_adjoinable*) means that it can occur without being selected by a head. Thus, VP is not adjoinable because it must be selected by v.

moved. If the targeted item is a primitive head, then there is an extra concern: how much of the surrounding structure should come with the head?

```
def externalize_structure(self, ps):
    if ps.head().is_adjoinable():
        if ps.is_complex():
            self.externalize(ps)
        else:
            self.externalize_head(ps, ps.tail_test())
```

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

```
def externalize_head(self, head, tail_test):
    if (tail_test and '!SPEC:*' in head.features and head.edge()) or (not tail_test and
self.capture_specifier_rule(head)):
        self.externalize(head.mother.mother) # Externalize with specifier
    else:
        self.externalize(head.mother) # Externalize without specifier

def capture_specifier_rule(self, head):
    return head.edge() and '-ARG' not in head.features and head.mother.mother and '-SPEC:*' not in
head.features and \
        not (set(head.specifiers_not_licensed()) & set(next((const for const in head.edge()),
None).head().features))
```

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

```
def externalize(self, ps):
    [...]
    ps.adjunct = True
    self.add_tail_features_if_missing(ps)
    self.transfer_adjunct(ps)
    return True
```

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

7.3 Agreement reconstruction Agree-1 (agreement_reconstruction.py)

Agreement reconstruction uses minimal search to find heads with feature [VAL], and then values all unvalued features of those heads, if any. This valuation is called Agree-1, or agreement reconstruction.

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

Agree-1 examines the local domain surrounding the head and detects any referential or other arguments within that domain. It examines the complement first, and then if unvalued features remain, the edge.


```

def Agree_1(self, head):
    if head.sister():
        goal, phi = self.Agree_1_from_sister(head)
        if phi:
            [...]
            for p in phi:
                self.value(head, goal, p, 'sister')
            if not is_unvalued(head):
                return

    goal2, phi = self.Agree_1_from_edge(head)
    if goal2:
        for p in phi:
            if {f for f in head.features if unvalued(f) and f[:-1] == p[:len(f[:-1])]}:
                self.value(head, goal2, p, 'edge')
    [...]

```

Notice that if unvalued features do not remain after the algorithm has searched the sister, the operation ends. The “functional motivation” behind valuation is that it pairs the predicate with its argument, where a predicate is a lexical item with unvalued phi-features and the argument is a phrase headed by a lexical item with valued phi-features. Agreement from sister is defined as follows.

```

def Agree_1_from_sister(self, head):
    #=====
    for node in head.sister():
        if node.left_const:
            if node.left_const.is_complex() and self.agreement_condition(head, node.left_const):
                return node.left_const.head(), sorted({f for f in node.left_const.head().features if
                    phi(f) and f[:7] != 'PHI:DET' and valued(f)})
            else:
                break
    return None, None
    #=====

```

It executes minimal search into the sister and locates complex left phrases whose head contains valued phi-features (function `agreement_condition()`), and then returns the head and the phi-features. The actual valuation will be handled inside the caller (function `Agree_1()` above). If the left node is primitive, search is terminated, so it is confined into local domain and does not look “past” any other head, barring long-distance agreement.¹⁷ Agreement from the edge uses the path mechanism:

```

def Agree_1_from_edge(self, head):
    return next((const.head(), sorted({f for f in const.head().features if phi(f) and valued(f)}))
        for const in [const for const in head.edge()] + [head.extract_pro()] if
        const and self.agreement_condition(head, const)), (None, {}))

```

Here we locate the most local referential phrase (phrase with ϕ -features) from the edge or, if none is found, we examine if there is a virtual pro-element in the head (function `extract_pro()`). Valuation is the actual operation which fills in the unvalued slots with the valued counterparties.

```

def value(self, h, goal, phi, location):
    if h.get_valued_features() and self.valuation_blocked(h, phi):
        h.features.add(mark_bad(phi))
    if {f for f in h.features if unvalued(f) and f[:-1] == phi[:len(f[:-1])]}:
        h.features = h.features - {f for f in h.features if unvalued(f) and f[:-1] == phi[:len(f[:-1])]}

```

¹⁷ This assumption most likely requires some revision in the future.

```

h.features.add(phi)
[...]
h.features.add('PHI_CHECKED')

```

The function checks first that there are no feature conflicts. This occurs if we attempt to value feature with some type *T* and value *V* against a head that already has a different specification for the same type, leading into what we would consider an “agreement error.” Next, the incoming valued feature (*phi*) will be added to the head while the corresponding unvalued counterpart is deleted. Notice that an unvalued feature will always end with `_`. The feature `[PHI_CHECKED]` will be added to the head, which signals that some *phi*-features were checked during Agree-1.

There are several controversial and nontrivial aspects of these operations. One is that agreement with a head and a referential argument inside its sister is registered semantically as a predicate-argument dependency.

```

self.brain_model.narrow_semantics.predicate_argument_dependencies.append((head, goal))

```

This is used later in certain computations, but it expresses the intuition that the function of agreement reconstruction is to pair predicates with arguments. There is also a special condition which prevents the valuation of definiteness feature from the sister, and on such basis requires that this is valued from the edge.

```

if node.left_const.is_complex() and self.agreement_condition(head, node.left_const):
    return node.left_const.head(), sorted({f for f in node.left_const.head().features if phi(f) and
f[:7] != 'PHI:DET' and valued(f)})

```

This captures the idea that the EPP condition is related to the valuation of *D*, as if *D* would be a “criterial feature for A-reconstruction,” at least in some languages, but the ultimate implementation remains mysterious. Therefore, I do not believe that this is the correct way to handle this issue, but now it is implemented here. Finally, the agreement condition contains a stipulation which establishes that Finish predicates only agree with nominative and genitive arguments. I do not know how to derive the desired results from any generalization, so the stipulation, which is questionable and most likely incorrect, still remains in this version.

7.4 LF-legibility (LF.py)

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

```

self.LF_legibility_tests = [self.selection_tests,
                             self.projection_principle,
                             self.head_integrity_test,
                             self.probe_goal_test,
                             self.semantic_complement_test,

```

```

self.double_spec_filter,
self.criterial_feature_test,
self.adjunct_interpretation_test]

```

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

```

def LF_legibility_test(self, ps, special_test_battery=None):
    if special_test_battery:
        self.active_test_battery = special_test_battery
    else:
        self.active_test_battery = self.LF_legibility_tests
    return self.pass_LF_legibility(ps)

```

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, ps):
    if ps.is_primitive():
        self.local_edge = next((const for const in ps.edge()), ps.extract_pro())
        for LF_test in self.active_test_battery:
            if not LF_test(ps):
                return False
    else:
        if not ps.left_const.find_me_elsewhere:
            if not self.pass_LF_legibility(ps.left_const):
                return False
        if not ps.right_const.find_me_elsewhere:
            if not self.pass_LF_legibility(ps.right_const):
                return False
    return True

```

Of special interest is the selection test, which examines if specifier and complement selection tests are valid. Selection tests are collected into their own data structure

```

self.selection_tests = [self.selection__negative_specifier,
                        self.selection__unselective_negative_specifier,
                        self.selection__unselective_negative_edge,
                        self.selection__negative_one_edge,
                        self.selection__positive_obligatory_complement,
                        self.selection__negative_complement,
                        self.selection__unselective_negative_complement,
                        self.selection__positive_unselective_complement,
                        self.selection__positive_obligatory_specifier,
                        self.selection__unselective_positive_specifier]

```

from where they are then applies to every lexical feature:

```

def selection_test(self, head):
    return next((test(head, lexical_feature) for lexical_feature in
sorted(for_lf_interface(head.features))
        for test in self.selection_tests if not test(head, lexical_feature))), True)

```

This function returns the truth value of the first failed test (False), or return True if all tests pass. The function next(), which finds the first test that does not pass, is used to speed up processing.

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 (`interpret_(ps)`). Then it performs two other global interpretation operations: assignment generation and generation of the information structure.

```
def postsyntactic_semantic_interpretation(self, root_node):
    [...]
    self.interpret_(root_node)
    self.quantifiers_numerals_denotations_module.reconstruct_assignments(root_node)
    self.pragmatic_pathway.calculate_information_structure(root_node, self.semantic_interpretation)
    self.document_interface_content_for_user()
    return not self.semantic_interpretation_failed
```

The recursive interpretation function examines each primitive lexical item and applies several interpretation operations to it.

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

All primitive lexical elements are targeted for interpretation; complex phrases are recursed. Primitive lexical elements are subjected to LF-recovery (Section 4.10.4), phi-conflict detection, tail feature interpretation, inventory projection (Section 4.10.1), operator binding (Section 4.10.6) and pragmatic processing (4.10.5). The general idea is that lexical features are diverged into different semantic subsystems for interpretation. This is not psycholinguistically realistic, in that semantic interpretation is generated phase-by-phase basis but, as explained in Section 4.10.2, incremental interpretation requires semantic backtracking.

7.5.2 *Projecting semantic inventories (semantic switchboard)*

One function of narrow semantics is to project semantic objects, corresponding to the expressions in the input sentence, into the semantic inventories. This is done by function `inventory_projection()`. Suppose we are examining lexical item α . If α has lexical features that can be interpreted by one or several of the semantic subsystems, narrow semantics queries the corresponding system and, if that system can process that lexical feature, asks it to project the corresponding semantic object into existence and then links these objects to the original expression by using a lexical referential index feature. The referential index feature can be thought of as a link between the expression and the corresponding semantic object. It has form `[IDX:N,S]` where N is a numerical identifier and S denotes the semantic space.

```
def inventory_projection(self, ps):
    def preconditions(ps):
```

```

        return not self.controlling_parsing_process.first_solution_found and \
            not ps.find_me_elsewhere and \
            'BLOCK_NS' not in ps.features

    if preconditions(ps):
        for space in self.semantic_spaces:
            if self.query[space]['Accept'](ps.head()):
                idx = str(self.global_cognition.consume_index())
                ps.head().features.add('IDX:' + idx + ',' + space)
                self.query[space]['Project'](ps, idx)
                self.query[space]['Denotation'] = \
                    self.query['GLOBAL']['Project'](ps,
self.transform_for_global_inventory(self.query[space]['Get'](idx))
                if space == 'QND':
                    ps.head().features.add('REF')
            ps.features.discard('BLOCK_NS')

```

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

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

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

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

then asks the subsystem to project the corresponding element to the semantic inventory.

Properties of the projected item are determined by the lexical features in α . Corresponding projection to the global inventory space will also take place.¹⁸ 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 into its own semantic inventory that get linked with referential expressions occurring in phrase

¹⁸ The assumption that every referential expression projects an entity into the discourse inventory might sound odd, since in many cases they should refer to an existing object instead. For example, pronoun *he* will typically denote an existing object. Indeed, it might. However, each time a referential expression is encountered in the input a new object is always projected, as shown by the code above, regardless of whether it will in the end be selected as a likely denotation

structure objects at the syntax-semantics interface. It can also understand and interpret referential lexical features such as ϕ -features and translate them into semantic features.

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

```
def reconstruct_assignments(self, ps):
    self.referential_constituents_feed = self.calculate_possible_denotations_(ps)
    self.create_assignments_from_denotations_(0, 0, {})
    self.narrow_semantics.semantic_interpretation['Assignments'] = self.all_assignments
```

The first line calculates possible denotations for all relevant referential expressions in α and returns a list of expressions [EXP₁, EXP₂, ...] for which this operation was performed. The list is used to make subsequent processing easier and has no cognitive role in the theory. The denotations are stored as denotation sets inside QND space semantic objects (that is, inside this module itself). Denotation sets contain pointers to objects in the global discourse space. For example, if the original expression is pronoun *he*, then the QND space entry may contain a set of denotations {1, 2}, where the numbers refer to two persons ‘John₁’ and ‘Simon₂’ in the global discourse inventory. Once this is done, the function calls recursive assignment function *create_assignments_from_denotations*. All logically possible assignments are considered. They are not obviously calculated during real-time language processing; rather, they are part of linguistic competence. The assignments are stored into a QND-internal data structure.

Let us examine the details. Possible denotations are calculated by the following function.

```
def calculate_possible_denotations_(self, ps):
    L1 = []
    L2 = []
    if not ps.find_me_elsewhere:
        if ps.is_complex():
            L1 = self.calculate_possible_denotations_(ps.left_const)
            L2 = self.calculate_possible_denotations_(ps.right_const)
        else:
            if self.narrow_semantics.has_referential_index(ps, 'QND'):
                idx, space = self.narrow_semantics.get_referential_index_tuples(ps, 'QND')
                self.inventory[idx]['Denotations'] = self.create_all_denotations(ps)
                return [(idx, f'{ps.illustrate()}', ps, self.inventory[idx]['Denotations'])]
    return L1 + L2
```

The first parts are involved with recursion. We can look at the *else*-clause. A lexical item that has a referential feature linking it with a semantic object in the QND space (*has_referential_index()*) will be provided with denotations by *create_all_denotations()*. The code returns a list of tuples $\langle \text{idx}, \text{constituent printout}, \text{constituent}, \text{denotations} \rangle$, so that the whole recursion will return a longer list containing all expressions that were provided with this information. This list, which is a simple auxiliary representation that plays no role in the theory, will then be used to build the assignments. Function *create_all_denotations()* will return a set of denotations (global inventory objects) that

satisfy the criteria stored in the QND entry. For example, a pronoun *he* can only pick singular male objects, and so on.

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

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

```
def create_assignments_from_denotations(self, c_index, d_index, one_complete_assignment):
    idx, const, ps, denotations = self.referential_constituents_feed[c_index]
    denotation = denotations[d_index]
    one_complete_assignment[idx] = denotation
    if len(one_complete_assignment) == len(self.referential_constituents_feed):
        self.all_assignments.append(self.calculate_assignment_weight(one_complete_assignment))
    if c_index < len(self.referential_constituents_feed) - 1:
        self.create_assignments_from_denotations(c_index + 1, 0, one_complete_assignment.copy())
    if d_index < len(denotations) - 1:
        self.create_assignments_from_denotations(c_index, d_index + 1,
            one_complete_assignment.copy())
```

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

```
def calculate_assignment_weight(self, complete_assignment):
    weighted_assignment = complete_assignment.copy()
    weighted_assignment['weight'] = 1
    for expression in self.referential_constituents_feed:
        if not self.binding_theory_conditions(expression, complete_assignment):
            weighted_assignment['weight'] = 0
        if not self.predication_theory_conditions(expression, complete_assignment):
            weighted_assignment['weight'] = 0
    return weighted_assignment
```

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

Assignments are first filtered by conditions that mimic the binding conditions A-C, function `binding_theory_conditions()` above. The general idea is that referential expressions can contain grammaticalized features that function as “instructions” for a system that knocks out logically possible assignments. This filtering is performed by the following function, which in effect incorporates the binding theory.

```

def binding_theory_conditions(self, expression, complete_assignment):
    idx, name, ps, denotations = expression
    for feature in list(self.get_R_features(ps)):
        D, rule, intervention_feature = self.open_R_feature(feature)
        if {rule} & {'NEW', 'OLD'}:
            reference_set = self.reference_set(ps, intervention_feature, complete_assignment)
            if not
self.narrow_semantics.global_cognition.general_evaluation(complete_assignment[idx], rule,
reference_set):
    return False
    return True

```

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

```

def reference_set(self, ps, intervention_feature, complete_assignment):
    return {complete_assignment[self.narrow_semantics.get_referential_index(head, 'QND')]
            for head in ps.constituent_vector(intervention_feature)
            if self.narrow_semantics.has_referential_index(head) and
            self.narrow_semantics.exists(head, 'QND')}

```

We pick up all semantic objects accessed by heads inside a constituent vector (upward path, see Section 7.1.3) from α . General evaluation, which is part of global cognition, is provided by the following operation that performs a simple set-theoretical comparisons.

```

def general_evaluation(self, mental_object, rule, reference_set):
    if 'NEW' in rule:
        return not {mental_object} & reference_set
    if 'OLD' in rule:
        return {mental_object} & reference_set

```

The intuition is that narrow semantics has direct access to the syntax-semantic interface objects and to the general evaluation operation.

To illustrate these operations by using a concrete example, consider the processing of a simple pronoun *he* that is inside a larger expression $\alpha = \dots he \dots$. All referential expressions in α , including the pronoun, are first linked with a set of possible denotations. These sets depend on what entities exists in the global discourse inventory at the time the operation takes place, hence the process takes place at the language-cognition interface. Suppose we have three male persons John₁, Simon₂ and an unknown third person₃ that was projected by default when *he* was first interpreted in the global discourse inventory. The pronoun will be associated with the set {1, 2, 3}, because it could refer to any of these three entities. The system will then consider all possible assignments and select the ones that are most likely and/or plausible, given the context and other

factors. Binding theory restricts these assignments. Assignments like *Simon₂ admires him₂* and *Simon₃ admires him₃* are both ruled out, because the pronoun would have “too local” antecedent.

7.5.4 Operator-variable interpretation (*SEM_operators_variables.py*)

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

```
def bind_operator(self, head, semantic_interpretation):
    if not self.scope_marker(head):
        for operator_feature in [feature for feature in head.features if
            self.is_operator_feature(feature)]:
            scope_marker_lst = self.bind_to_propositional_scope_marker(head, operator_feature)
            if len(scope_marker_lst) > 0:
                semantic_interpretation['Operator bindings'].append((f'{head.illustrate()}', f'by
                    {scope_marker_lst[0]}[{operator_feature}]'))
                self.interpret_operator_at_lexical_item(operator_feature, head,
                    semantic_interpretation)
                [...]
            else:
                self.interpretation_failed = True
                break
```

The first line ignores reflexive scoping where the scope marker {FIN, OP} is interpreted as defining the scope for itself. Next we examine all operator features inside the lexical item and bind each operator with its scope marker. If no scope marker is found, interpretation is marked as a failure. If it is found, the results are stored for later use, and an additional interpretation function is applied. This function processes the lexical content of the operator (verum focus interpretation, complex predicates as operators). Binding is calculated by

```
def bind_to_propositional_scope_marker(self, head, operator_feature):
    local_mandatory_binder = next((node for node in head.working_memory_path() if {operator_feature,
        'FIN'}.issubset(node.features)), None)
    if not local_mandatory_binder:
        if 'OVERT_SCOPE' not in head.features:
            return [node for node in head.working_memory_path() if {'T',
                'FIN'}.issubset(node.features) or {'C', 'FIN'}.issubset(node.features)]
        else:
            return []
    return [local_mandatory_binder]
```

where we first attempt to locate the closest mandatory scope marker, as defined by the combination of finiteness and the operator itself, and then if none is found, consider finite boundaries provided that overt scope is not required.

The design philosophy follows Chomsky’s duality of semantic interpretation hypothesis (Chomsky 2008). The idea is that the operator-variable module that is part of narrow semantics “understands” operator variable constructions and is able to feed the information for other cognitive systems. Intuitively the operation creates unsaturated properties from saturated propositions by abstracting over a variable. The result is an interpretation where, instead of focusing on one saturated proposition, we have either a completely open set of propositions, as in the case of relative clauses, or a finite set of propositions, as in the case of various contrastive

focus constructions. The result is a situation in which the proposition is interpreted against a background set of other propositions.

7.5.5 *Pragmatic pathway*

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

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

```
self.pragmatic_pathway.interpret_discourse_features(ps, self.semantic_interpretation))
```

This function will handle all discourse features.

```
def interpret_discourse_features(self, ps, semantic_interpretation):
    self.refresh_inventory(ps)
    d_features = self.get_discourse_features(ps.features)
    for f in sorted(d_features):
        result = self.interpret_discourse_feature(f, ps)
        if not result:
            return []
    semantic_interpretation['DIS-features'].append(result)
```

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

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

```
self.pragmatic_pathway.calculate_information_structure(ps, self.semantic_interpretation)
```

```
def calculate_information_structure(self, ps, semantic_interpretation):
    if 'FIN' in ps.head().features:
```

```

semantic_interpretation['Information structure'] =
self.create_topic_gradient(self.arguments_of_proposition(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:

```

self.controlling_parser_process.narrow_semantics.pragmatic_pathway.allocate_attention(terminal_lexical_item)

```

This line 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 {'D', 'φ', 'P'} & head.features:
        idx = self.consume_index()
        head.features.add('IDX:'+str(idx))
        self.records_of_attentional_processing[str(idx)] = {'Order':idx, 'Name': f'{head}'}

```

The first line determines what kind of elements are included into the attentional mechanism, and depends on the interests of the researcher. Then, some attentional resources are allocated to the processing, and order information is stored. Finally, when the sentence comes through the LF-interface, the pragmatic module attempts to construct the topic gradient on the basis of this and other sources of information. The presentation order at which linguistic information was originally presented and processed, as shown above, is now interpreted as representing relative topicality.¹⁹

Topic gradient calculations are nontrivial for several reasons. First, they must ignore some noncanonical word order changes, such as those created by \bar{A} 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

¹⁹ 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.

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}
```

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 line

```
self.controlling_parser_process.narrow_semantics.pragmatic_pathway.unexpected_order_occurred(dropped_
floater, starting_point_head)
```

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.

7.5.6 *LF-recovery*

LF-recovery is triggered if an unvalued phi-feature occurs at the LF-interface. It is understood as a last resort operation, where the working memory path is explored for a possible antecedent. The operation returns a list of possible antecedents for the triggering head which are then provided in the results and log files.

```
def perform_LF_recovery(self, head, semantic_interpretation_dict):
    unvalued = must_be_valued({f for f in head.features if f[:4] == 'PHI:' and f[-1] == '_'})
```

```

        if unvalued:
            self.interpret_antecedent(self.LF_recovery(head, unvalued), head, unvalued,
semantic_interpretation_dict)

```

The function `LF_recovery` finds the possible antecedents, and function `interpret_antecedent` provide verbal feedback for the user on the basis of the former. If the lexical item contains unvalued number and person agreement features, then standard control dependency is activated.

```

def LF_recovery(self, probe, unvalued_phi):
    [...]
    if 'PHI:NUM:' in unvalued_phi and 'PHI:PER:' in unvalued_phi:
        if probe.is_primitive() and probe.is_left() and probe.sister().is_complex():
            working_memory.append(probe.sister())
            working_memory.extend(
                list(takewhile(self.recovery_termination, probe.working_memory_path()))
            )
            antecedent = next(
                (const for const in working_memory if self.is_possible_antecedent(const, probe)), None)
    [...]

```

The function works by creating a list of possible antecedents into the list data structure `working_memory` and then selects the closest antecedent from that list. The initial list contains the sister of the probe plus what is available in the working memory path (22). The `takewhile` function collects elements from the working memory path until `recovery_termination` condition is true, which captures certain locality properties. This list is then used to find possible antecedents, and the most local antecedent is selected and returned to the caller. A possible antecedent is one which is able to value all valued phi-features of the probe:

```

def is_possible_antecedent(self, antecedent, head):
    unchecked = get_semantically_relevant_phi(head)
    for F in antecedent.head().get_valued_features():
        for G in get_semantically_relevant_phi(head):
            unchecked = check(F, G, unchecked)
    if not unchecked:
        return True

```

The function checks features from the initial list of unchecked features and returns True only if none remains. Once the list of antecedents has been gathered and returned to this caller function, the results are interpreted. This function prints the results into the output so that they appear in the results file for the user.

```

def interpret_antecedent(self, list_of_antecedents, probe, unvalued, semantic_interpretation_dict):
    self.LF_recovery_results = set()
    if list_of_antecedents:
        self.LF_recovery_results.add(self.antecedent_exists(probe, list_of_antecedents[0]))
        self.brain_model.narrow_semantics.predicate_argument_dependencies.append((probe,
list_of_antecedents[0].head()))
        semantic_interpretation_dict['Recovery'].append(self.LF_recovery_results)
    else:
        self.LF_recovery_results.add(f'{probe}(' + self.antecedent_does_not_exist(probe, unvalued) +
')')
    self.report_to_log(probe, list_of_antecedents, unvalued)
    self.brain_model.consume_resources("LF recovery")
    self.brain_model.consume_resources("Phi")

```

The two functions `antecedent_exists` and `antecedet_does_not_exist` interpret situations where an antecedent was found (this is where we get the ‘agent’ and ‘patient’ interpretations) and not found (which generates e.g. clausal antecedent interpretations, generic interpretations), respectively.

Current definitions for agent and patient interpretations are such that patients are right sisters while everything else is interpreted as representing agent arguments. The label of the head of the antecedent is also taken into account, so that referential argument antecedents are interpreted differently than clausal arguments. Much of the contents of these functions are stipulative and aimed only for generating predictions for the user.

The recovery function has few lines of Finnish specific code, which is used to model finite control in Finnish. The operation is triggered if an unvalued definiteness feature remains in the lexical head, following a proposal by Anders Holmberg and colleagues.

```
def LF_recovery(self, probe, unvalued_phi):
    [...]
    if 'PHI:NUM:_' in unvalued_phi and 'PHI:PER:_' in unvalued_phi:
    [...]
    if 'PHI:DET:_' in unvalued_phi:
        # ----- minimal search -----
        for const in probe.working_memory_path():
            if self.special_local_edge_antecedent_rule(const, probe, list_of_antecedents):
                break
            if self.is_possible_antecedent(const, probe):
                list_of_antecedents.append(const)
        # -----
    [...]
    return list_of_antecedents
```

The essence of this mechanism is that it is nonlocal and explores the contents of the whole working memory path, resulting in the finite control profile. The special edge antecedent rule handles the mysterious EPP condition, whereby a phrasal constituent at the SpecTP terminates the search and triggers the generic reading. This should be unified with the general interpretation rule above, but it is now here since the whole system applies only Finnish. To examine the matter in detail, we need data from other languages exhibiting finite control.

8 Formalization of the parser

8.1 Linear phase parser (linear_phase_parser.py)

8.1.1 The brain model

The linear phase (LP) parser module `linear_phase_parser.py` defines the behavior of the parser. It defines an idealized “brain 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. The main script creates a separate brain model for speakers of any language present in the lexicon. These brain models differ only in terms of the composition of (some) lexical items; the computational core remains the same. We imagine the brain model as a container that hosts all modules and their connections, as shown in Figure 32.

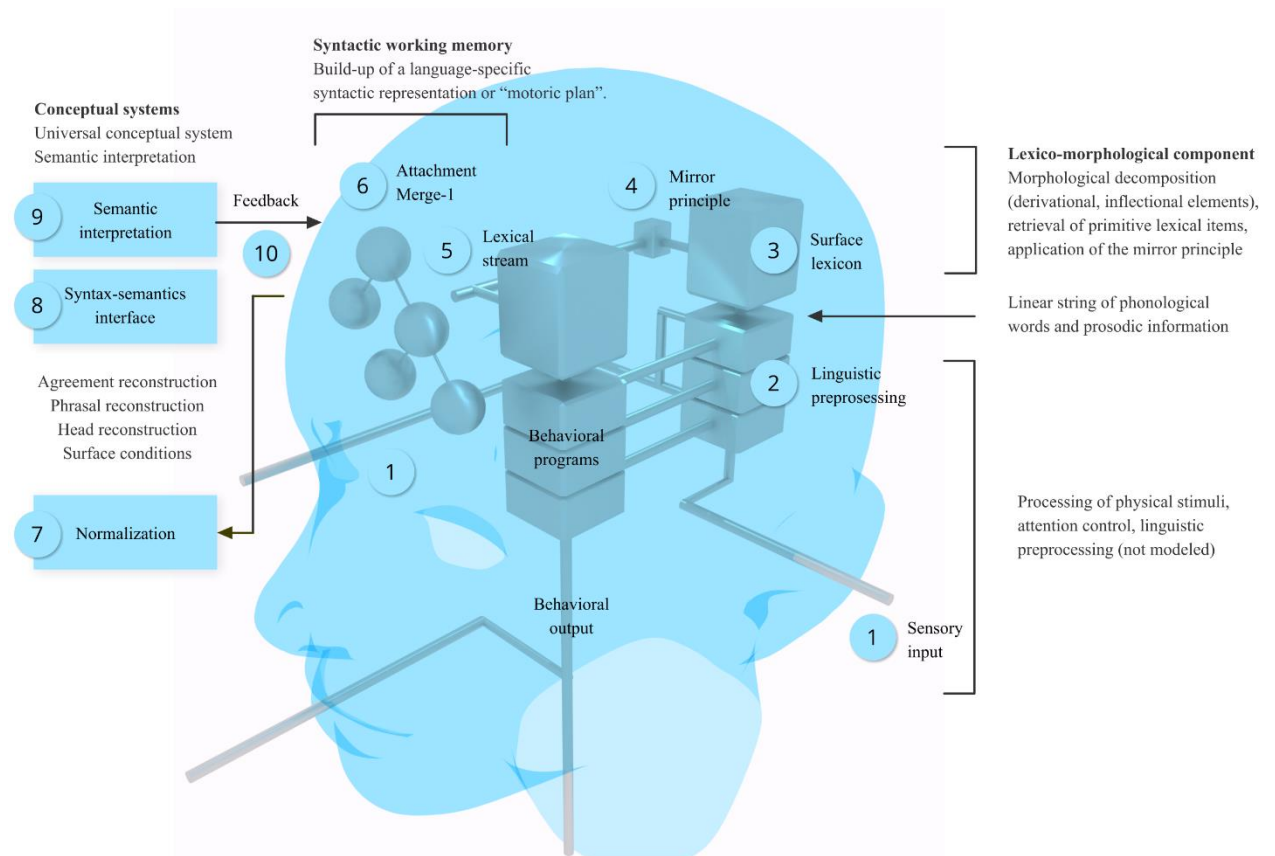


Figure 32. A brain model contains all submodules defined by the theory and their connections.

8.1.2 Recursive parsing function (*_first_pass_parse*)

When the main script wants to parse a sentence, it sends the sentence to the brain model (linear phase parser) as a list of words. The list must be tokenized correctly by the user. The parsing function is `parse()`. 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 `_first_pass_parse()` with three arguments *current structure*, *list* and *index*, with *current structure* being empty and *index* = 0. This function processes the whole list, creates a recursive parsing tree, and provides an output.

The recursive parsing function takes the currently constructed phrase structure α , a linearly ordered list of words and an index in the list of words as its arguments. It will first check if 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.

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

def circuit_breaker(self, ps, lst, index):

    # We have decided not to explore any more solutions, exit the recursion
    if self.exit:
        return True

    if index < len(lst):
        log(f'\n\t\tNext item: {lst[index]}. ')

    # If there are no more words, we attempt to complete processing
    if index == len(lst):
        self.complete_processing(ps)
        return True

    # Set the amount of cognitive resources (in ms) consumed based on word length
    self.time_from_stimulus_onset = int(len(lst[index]) * 10)

    # Add the time to total time if we haven't yet found any solutions
    if not self.first_solution_found:
        self.resources['Total Time']['n'] += self.time_from_stimulus_onset

    return False
```

If there are no more words, α will be send out for interpretation. This function is `complete_processing()`. It implements several operations (transfer, LF-legibility, semantic interpretation), discussed later. 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.

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

This operation corresponds to a stage in language comprehension in which an incoming sensory-based item is matched with a lexical entry in the surface vocabulary. If *w* is ambiguous, all corresponding lexical items will be returned as a list that will be explored in some order. The

ordered list will be added to the recursive loop as an additional layer. The lexical retrieval function (from the lexical interface class) is below, followed by few comments.

```
def lexical_retrieval(self, phonological_entry):
    [...]
    if phonological_entry in self.surface_vocabulary:
        word_list = [const.copy() for const in self.surface_vocabulary[phonological_entry]]
    [...]
    else:
        const = self.PhraseStructure()
        const.features = {'PF:?', '?'}
        const.morphology = phonological_entry
        const.internal = internal
        word_list = [const]
    [...]
    return word_list
```

The first line matches the phonological string with items in the surface lexicon and retrieves the lexical entries into a word list. If the incoming phonological string is unknown, an ad hoc lexical entry is created.

Here it becomes important to be aware of the exact technical implementation of the lexicon-syntax interface. The lexicon is a mapping from phonological surface strings (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 never 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 (e.g., /the/ ~ D), then the constituent, primitive lexical item, will contain the set of features as specified in the lexical entry. If the lexical entry maps into an inflectional morpheme, then it will again consist of a set of (inflectional) features, but these will be processed differently. The following figure illustrates the idea.

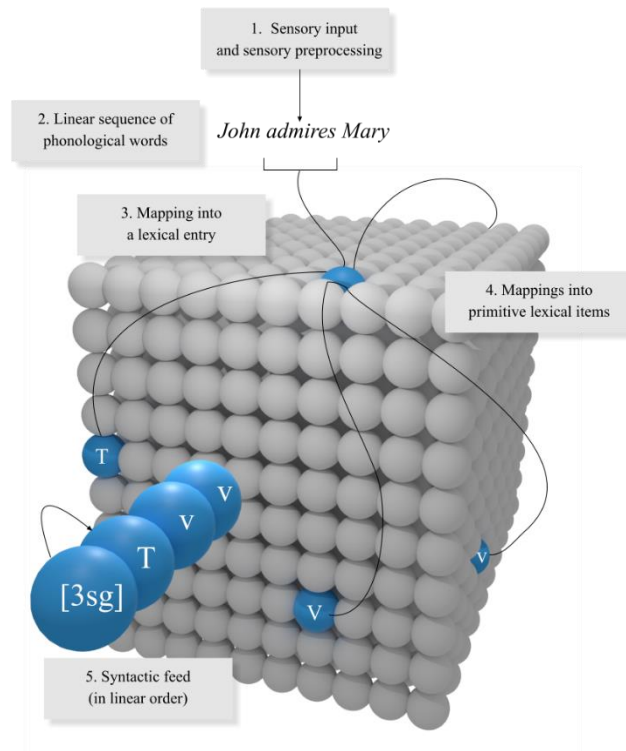


Figure 40. Organization of the lexicon. The lexicon is a mapping from phonological surface strings into constituents which make up the lexicon. Some surface strings map into primitive lexical items (blue items) which are sets of lexical features (e.g., /the/ ~ D). Other surface strings map into morphological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

This architecture implies that 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 returned by the lexical retrieval may still consists of a morphological decomposition (figure 40). The morphological parser will return a new list of words that 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.

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

It is important to realize that 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).

```
def morphological_parse(self, controlling_parsing_process, lexical_item, input_word_list, index):
    current_lexical_item = lexical_item
    while self.is_polymorphemic(current_lexical_item):
        [...]
        morpheme_list = self.decompose(current_lexical_item.morphology)
        [...]
        self.apply_mirror_principle(input_word_list, morpheme_list, index)
        [...]
        current_lexical_item = self.lexicon.lexical_retrieval(input_word_list[index])[0]

    inflection_features = self.get_inflection_features(current_lexical_item, input_word_list[index])

    return current_lexical_item, input_word_list, inflection_features
```

The first operation decomposes the lexical item into a morpheme list, if it is polymorphemic, which is then reversed by applying the mirror principle. The first morpheme is then used to retrieve a lexical item from the lexicon. Finally, the *first item* in the new list will be sent to the syntactic component via lexical stream.

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

The lexical stream pipeline handles several operations. For example, some of the morphemes could be inflectional, in which case they are stored as features into a separate memory buffer inside the lexical stream and then added to the next and hence also adjacent morpheme when it is being consumed in the reversed order. If inflectional features were encountered instead of a morpheme, the parsing function is called again immediately without any rank and merge operations. If there were several inflectional affixes, they would be all added to the next morpheme consumed from the input. Other lexical items enter the syntactic module.

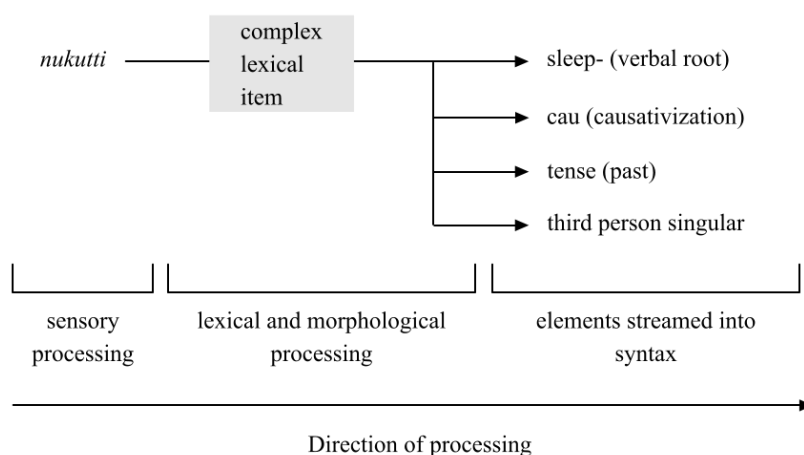


Figure 41. Lexical processing pipeline.

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.

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

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

```
for site, transfer in merge_sites:
    left_branch = self.target_left_branch(ps, site)
    new_constituent = self.attach(left_branch, site, terminal_lexical_item, transfer)
    self.put_rest_out_of_working_memory(merge_sites)
    self.parse_new_item(new_constituent, lst_branched, index + 1)
    if self.stop_looking_for_further_solutions():
        break
```

The loop examines each (site, transfer) pair provided by the plausibility function above. The sites are right edge nodes in the partial phrase structure currently being developed. Transfer is a Boolean variable telling whether we want to transfer the left branch or not. 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 Boolean transfer parameter controls whether the left branch of the new constituent will be transferred during Merge-1.

The code above contains two essential functions: `target_left_branch()` and `attach()`. The former is required because 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 phrasal Merge-1 and sinking (merging into a complex head, Sections 4.8.5, 7.1.2.4, 0) depending on the situation (below) and performs working memory operations.

```
def attach(self, left_branch, site, terminal_lexical_item, transfer):
    self.maintain_working_memory(site)
    if self.belong_to_same_word(left_branch, site):
        new_constituent = self.sink_into_complex_head(left_branch, terminal_lexical_item)
    else:
        new_constituent = self.attach_into_phrase(left_branch, terminal_lexical_item, transfer)

    if not self.first_solution_found:
        self.time_from_stimulus_onset_for_word.append((terminal_lexical_item,
self.time_from_stimulus_onset))

    return new_constituent
```

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

```

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

```

Currently these involve only conditions regulating clitics, which must observe certain properties regulating the syntax-phonology mapping. The issue is empirically nontrivial. If the surface conditions are satisfied, the whole phrase structure object will be transferred and then checked for LF-legibility and semantic interpretation:

```

if not self.LF.LF_legibility_test(ps) or not self.LF.final_tail_check(ps) or not
self.narrow_semantics.postsyntactic_semantic_interpretation(ps_):
    self.add_garden_path()

```

If these fail, the result is rejected; otherwise processing continues. The rest of the operations performed inside this function are (important) logging operations. Notice that the whole global semantic interpretation is interpreted by this line. 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.

```

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

```

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, ps, w):
    nodes_not_in_active_working_memory = []
    [...]
    if self.word_internal(ps, w) and self.dispersion_filter_active():
        solutions = [(ps.bottom(), True)]
    else:
        nodes_in_active_working_memory, nodes_not_in_active_working_memory =
self.in_active_working_memory(ps)
        nodes_available = self.filter(nodes_in_active_working_memory, w)
        merge_sites = self.rank_merge_right(nodes_available, w)
        all_merge_sites = merge_sites + nodes_not_in_active_working_memory
        solutions = self.evaluate_transfer(all_merge_sites)
    [...]
    return solutions

```

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 in the active working memory are processed by using filter and ranking; the rest are added to the solutions list in random order. Only nodes that pass the filter are ranked. Ranking uses several 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) $[\alpha \beta]$ can be rejected if it would break a configuration presupposed in word formation. (4) $[\alpha \beta]$ can be rejected if it constitutes an impossible sequence.

```
@knockout_filter
def filter(self, list_of_sites_in_active_working_memory, w):
    #-----geometrical minimal search-----
    for N in list_of_sites_in_active_working_memory:
        if self.does_not_accept_any_complementizers(N):
            log(f'Reject {N} + {w} because {N} does not accept complementizers...')
            self.brain_model.consume_resources('Filter solution')
            continue
        if N.is_complex() and self.left_branch_filter(N):
            log(f'Reject {N} + {w} due to bad left branch...')
            self.brain_model.consume_resources('Filter solution')
            continue
        if self.word_breaking_filter(N, w):
            log(f'Reject {N} + {w} because it breaks words...')
            self.brain_model.consume_resources('Filter solution')
            continue
        if self.impossible_sequence(N, w):
            log(f'Reject {N} + {w} because the sequence is impossible...')
            self.brain_model.consume_resources('Filter solution')
            continue
        adjunction_sites.append(N)
    #-----
    return adjunction_sites
```

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

```
def left_branch_filter(self, N):
    dropped, output_from_interfaces = self.brain_model.transfer_to_LF(N.copy())
    left_branch_passes_LF = self.brain_model.LF.LF_legibility_test(dropped,
self.left_branch_filter_test_battery)
    return not left_branch_passes_LF
```

8.2.3 Ranking (*rank_merge_right_*)

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

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

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

Each (site, weight) pair is 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 and evaluate whether they are true for this pair; if so, then the weight of α in the ranked list is modified according to the weight provided by the condition itself. The two nested loops implementing ranking are as follows:

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

In the current version the weight modifiers are ± 100 . These numbers outperform the small numbers assigned by the baseline weighting. They could compete on equal level if we assumed that the plausibility conditions provide smaller weight modifiers such as ± 1 . What the correct architecture is is an empirical matter that must be determined by psycholinguistic experimentation.

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

```
self.plausibility_conditions = \
    {'positive_spec_selection': {'condition': self.positive_spec_selection,
                                'weight':
self.controlling_parser_process.local_file_system.settings.get('positive_spec_selection', 100),
                                'log': '+Spec selection'},
    'negative_spec_selection': {'condition': self.negative_spec_selection,
                                'weight':
self.controlling_parser_process.local_file_system.settings.get('negative_spec_selection', -100),
                                'log': '-Spec selection'},
    'break_head_comp_relations': {'condition': self.break_head_comp_relations,
                                  'weight':
self.controlling_parser_process.local_file_system.settings.get('break_head_comp_relations', -100),
                                  'log': 'Head-complement word breaking condition'},
    'negative_tail_test': {'condition': self.negative_tail_test,
                           'weight':
self.controlling_parser_process.local_file_system.settings.get('negative_tail_test', -100),
                           'log': '-Tail'},
```

```

        'positive_head_comp_selection':    {'condition': self.positive_head_comp_selection,
                                           'weight':
self.controlling_parser_process.local_file_system.settings.get('positive_head_comp_selection', 100),
                                           'log': '+Comp selection'},
        'negative_head_comp_selection':    {'condition': self.negative_head_comp_selection,
                                           'weight':
self.controlling_parser_process.local_file_system.settings.get('negative_head_comp_selection', -100),
                                           'log': '-Comp selection'},
        'negative_semantics_match':        {'condition': self.negative_semantic_match,
                                           'weight':
self.controlling_parser_process.local_file_system.settings.get('negative_semantics_match', -100),
                                           'log': 'Semantic mismatch'},
        'lf_legibility_condition':         {'condition': self.lf_legibility_condition,
                                           'weight':
self.controlling_parser_process.local_file_system.settings.get('lf_legibility_condition', -100),
                                           'log': '-LF-legibility for left branch'},
        'negative_adverbial_test':         {'condition': self.negative_adverbial_test,
                                           'weight':
self.controlling_parser_process.local_file_system.settings.get('negative_adverbial_test', -100),
                                           'log': '-Adverbial condition'},
        'positive_adverbial_test':         {'condition': self.positive_adverbial_test,
                                           'weight':
self.controlling_parser_process.local_file_system.settings.get('positive_adverbial_test', 100),
                                           'log': '+Adverbial condition'}
    }

```

8.2.4 Knocking out heuristic principles

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

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

The number of various types of computational operations (e.g., Merge, Move, Agree) are also counted. The way they are counted merits a comment. Grammatical operations are counted as “black boxes” in the sense that we ignore all internal operations (e.g., minimal search, application of merge, generation of rejected solutions). The number of head reconstructions, for example, is increased by one if and only if a head is moved from a starting position X into a final position Y;

all intermediate positions and rejected solutions are ignored. This therefore quantifies the number of “standard” head reconstruction operations – how many times a head was reconstructed – that have been implemented during the processing of an input sentence. The number of all computational steps required to implement the said black box operation is always some linear function of that metric and is ignored. For example, countercyclic merge operations executed during head reconstruction will not show up in the number of merge operations; they are counted as being “inside” one successful head reconstruction operation. It is important to keep in mind, though, that each transfer operation will potentially increase the number independently of whether the solution was accepted or rejected. For example, when the left branch α is evaluated during $[\alpha \beta]$, the operations are counted irrespective of whether α is rejected or accepted during the operation.

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

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

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

If 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.

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

Thus, the user can add bookkeeping entries by adding the required key to the dictionary and then adding the line `controlling_parsing_process.consume_resources("key")` into the appropriate place

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

References

- Baker, Mark. 1996. *The Polysynthesis Parameter*. Oxford: Oxford University Press.
- Brattico, Pauli. 2016. “Is Finnish Topic Prominent?” *Acta Linguistica Hungarica* 63:299–330.
- Brattico, Pauli. 2018. *Word Order and Adjunction in Finnish*. Aarhus: Aguilu & Celik.
- Brattico, Pauli. 2020. “Finnish Word Order: Does Comprehension Matter?” *Nordic Journal of Linguistics* 44(1):38–70.
- Brattico, Pauli. 2021a. “A Dual Pathway Analysis of Information Structure.” *Lingua* 103156.
- Brattico, Pauli. 2021b. “Binding and the Language-Cognition Interfaces.” *In Preparation*.
- Brattico, Pauli. 2022. “A Multifeatural Path Analysis of Finnish Structural Case Assignment.” *Submitted*.
- Brattico, Pauli and Cristiano Chesi. 2020. “A Top-down, Parser-Friendly Approach to Operator Movement and Pied-Piping.” *Lingua* 233:102760.
- Chomsky, Noam. 1964. *Current Issues in Linguistic Theory*. Mouton.
- Chomsky, Noam. 1965. *Aspects of the Theory of Syntax*. Cambridge, MA.: MIT Press.
- Chomsky, Noam. 1981. *Lectures in Government and Binding: The Pisa Lectures*. Dordrecht: Foris.
- Chomsky, Noam. 1995. *The Minimalist Program*. Cambridge, MA.: MIT Press.
- Chomsky, Noam. 2000. “Minimalist Inquiries: The Framework.” Pp. 89–156 in *Step by Step: Essays on Minimalist Syntax in Honor of Howard Lasnik*, edited by R. Martin, D. Michaels,

- and J. Uriagereka. Cambridge, MA.: MIT Press.
- Chomsky, Noam. 2001. "Derivation by Phase." Pp. 1–37 in *Ken Hale: A Life in Language*, edited by M. Kenstowicz. Cambridge, MA.: MIT Press.
- Chomsky, Noam. 2005. "Three Factors in Language Design." *Linguistic Inquiry* 36:1–22.
- Chomsky, Noam. 2008. "On Phases." Pp. 133–66 in *Foundational Issues in Linguistic Theory: Essays in Honor of Jean-Roger Vergnaud*, edited by C. Otero, R. Freidin, and M.-L. Zubizarreta. Cambridge, MA.: MIT Press.
- Jelinek, Eloise. 1984. "Empty Categories, Case and Configurationality." *Natural Language & Linguistic Theory* 2:39–76.
- Kayne, Richard. 1983. "Connectedness." *Linguistic Inquiry* 14.2:223–49.
- Kayne, Richard. 1984. *Connectedness and Binary Branching*. Foris.
- Phillips, Colin. 1996. "Order and Structure." Cambridge, MA.
- Phillips, Colin. 2003. "Linear Order and Constituency." *Linguistic Inquiry* 34:37–90.