Computational implementation of a linear phase parser. Framework and technical documentation
(version 14.2)

2019

(Revised January 2023)

Pauli Brattico

Abstract

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

How to cite this document: Brattico, P. (2019). Computational implementation of a linear phase parser. Framework and technical documentation (version 14.2). IUSS, Pavia.

1 INTRODUCTION 6

2 INSTALLATION AND USE 7

2.1 INSTALLATION 7

2.2 USE 8

2.3 STRUCTURE OF THE SCRIPT 10

2.4 REPLICATION 10

2.5 DEFAULT TEST CORPUS 11

3 FRAMEWORK 12

3.1 THE FRAMEWORK 12

3.2 COMPUTATIONAL LINGUISTICS METHODOLOGY 15

3.3 OVERVIEW OF THE HYPOTHESIS 16

4 THE KERNEL (COMPREHENSION CYCLE) 21

4.1 INTRODUCTION 21

4.2 MERGE-1 21

4.3 LEXICAL SELECTION FEATURES 24

4.4 PHASES AND LEFT BRANCHES 26

4.5 LABELING 26

4.6 UPWARD PATHS AND MEMORY SCANNING 28

4.7 ADJUNCT ATTACHMENT 29

4.8 TRANSFER 32

4.8.1 Introduction 32

4.8.2 Phrasal reconstruction: general comments 32

4.8.3 Ā-chains 34

4.8.4 A-chains and EPP 36

4.8.5 Head reconstruction 37

4.8.6 Adjunct reconstruction 39

4.8.7 Minimal search 40

4.8.8 Agree-1 40

4.8.9 Ordering of operations 43

4.9 EPP AND THE EDGE FEATURE 44

4.10 LEXICON AND MORPHOLOGY 45

4.10.1 From phonology to syntax 45

4.10.2 Lexical items and lexical redundancy rules 45

4.10.3 Derivational and inflectional morphemes 46

4.11 NARROW SEMANTICS 46
4.11.1 Syntax, semantics and cognition 46
4.11.2 Argument structure 49
4.11.3 Antecedents and control 51
4.11.4 The pragmatic pathway 52
4.11.5 Operators 53
4.11.6 Binding 53

5 PERFORMANCE 55

5.1 HUMAN AND ENGINEERING PARSERS 55
5.2 MAPPING BETWEEN THE ALGORITHM AND BRAIN 55
5.3 COGNITIVE PARSING PRINCIPLES 56
5.3.1 Incrementality 56
5.3.2 Connectness 59
5.3.3 Seriality 59
5.3.4 Locality preference 59
5.3.5 Lexical anticipation 60
5.3.6 Left branch filter 60
5.3.7 Working memory 60
5.3.8 Conflict resolution and weighting 61
5.4 MEASURING PREDICTED COGNITIVE COST OF PROCESSING 61
5.5 A NOTE ON IMPLEMENTATION 62

6 INPUTS AND OUTPUTS 63

6.1 INSTALLATION AND USE 63
6.2 GENERAL ORGANIZATION 65
6.3 STRUCTURE OF THE INPUT FILES 67
6.3.1 Test corpus file (any name) 67
6.3.2 Lexical files (lexicon.txt, ug_morphemes.txt, redundancy_rules.txt) 69
6.3.3 Lexical redundancy rules 70
6.3.4 Study parameters (config_study.txt) 71
6.4 STRUCTURE OF THE OUTPUT FILES 71
6.4.1 Results 71
6.4.2 The log file 72
6.4.3 Simple logging 73
6.4.4 Saved vocabulary 73
6.4.5 Images of the phrase structure trees 74
6.4.6 Resources file 75
6.4.7 Semantic interpretation 75

6.4.8 Control and thematic roles ... 75

6.5 MAIN SCRIPT ... 76

7 GRAMMAR FORMALIZATION ... 78

7.1 BASIC GRAMMATICAL NOTIONS (PHRASE_STRUCTURE.PY) ... 78

7.1.1 Introduction ... 78

7.1.2 Lexical items ... 78

7.1.3 Phrase structure geometry ... 78

7.1.4 Complex heads and affixes ... 79

7.1.5 Sisters ... 79

7.1.6 Proper selected complement ... 80

7.1.7 Heads (labels), maximal projection, container ... 80

7.1.8 Minimal search, geometrical minimal search and upstream search ... 81

7.1.9 Upward paths ... 82

7.1.10 Cyclic Merge ... 82

7.1.11 Countercyclic Merge-1 ... 83

7.1.12 Remove ... 84

7.1.13 Detachment ... 84

7.1.14 Feature checking ... 84

7.1.15 Probe-goal: probe(label, goal_feature) ... 85

7.1.16 Tail-head relations ... 85

7.1.17 Abstractions ... 86

7.2 TRANSFER (TRANSFER.PY) ... 87

7.2.1 A comment on the current version (14.2) ... 87

7.2.2 Generalized transfer ... 88

7.2.3 Chain creation (all chains) ... 88

7.2.4 Head chain ... 90

7.2.5 A-chains ... 90

7.2.6 Ā-chains ... 90

7.2.7 Agree ... 90

7.2.8 Scrambling (scrambling_reconstruction.py) ... 91

7.2.9 Adjunct promotion (adjunct_constructor.py) ... 91

7.3 LF-LEGIBILITY (LF.PY) ... 92

7.3.1 Overall ... 92

7.3.2 Edge selection ... 93

7.3.3 Complement selection ... 94

7.3.4 Complex selection, projection principle ... 95

7.4 SEMANTICS (NARROW_SEMANTICS.PY) ... 95

7.4.1 Introduction 95

7.4.2 Projecting semantic inventories (semantic switchboard) 96

7.4.3 Quantifiers-numerals-denotations module 97

7.4.4 Operator-variable interpretation (SEM_operators_variables.py) 100

7.4.5 Pragmatic pathway 101

7.4.6 LF-recovery....... 103

8 FORMALIZATION OF THE PARSER 105

8.1 LINEAR PHASE PARSER (LINEAR_PHASE_PARSER.PY) 105

8.1.1 The brain model 105

8.1.2 Parse sentence (parse_sentence) 106

8.1.3 Recursive parsing function (parse_new_item)....... 106

8.2 PSYCHOLINGUISTIC PLAUSIBILITY 110

8.2.1 General architecture 110

8.2.2 Filtering....... 110

8.2.3 Ranking (rank_merge_right_) 111

8.2.4 Knocking out heuristic principles 113

8.3 RESOURCE CONSUMPTION 113

1 Introduction

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

This document describes properties of the version 14.2, which keeps within the framework of the original work but provides improvements, corrections and additions.[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 completed 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 gaps awaiting documentation. This happens when the material is still being worked on and has not been accepted for publication.

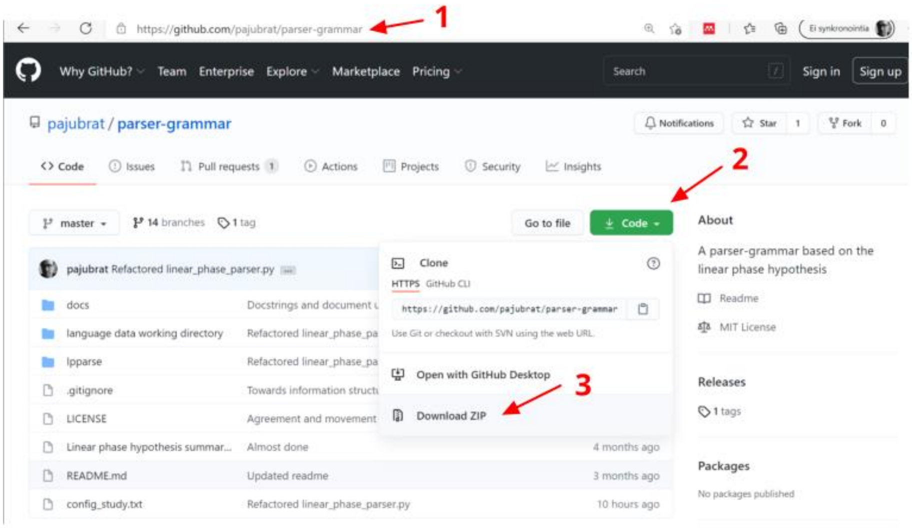
[1] The research was conducted in part under the research project "ProGraM-PC: A Processing-friendly Grammatical Model for Parsing and Predicting Online Complexity" funded internally by IUSS (Pavia).

[2] The model has been applied to filler-gap dependencies, Ā-reconstruction and pied-piping, local and nonlocal head movement, word order and adjunction, control and agreement, case assignment, information structure, binding, clitics, syntactic working memory, structural case assignment and the extended subject behavior (EPP). This document does not substitute for a proper scientific treatment of the topics covered. 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.

2 Installation and use

2.1 Installation

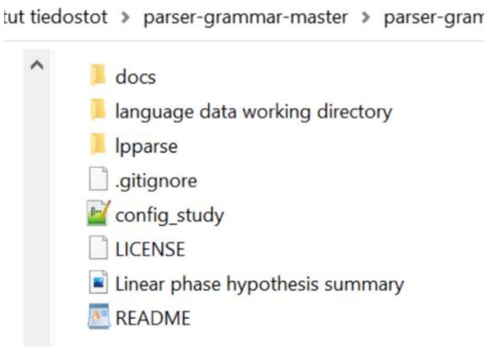
The linear phase comprehension algorithm is a collection of Python functions that processes natural language sentences by using principles of linguistic competence and performance. It works by reading test sentences from a dataset (test corpus) file and analyzing them. The program can be installed on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



pajubrat Refactored linear_phase_parser.py

docs	Docstrings and document u	
language data working directory	Refactored linear_phase_pa	
lpparse	Refactored linear_phase_pa	
.gitignore	Towards information structu	
LICENSE	Agreement and movement	
Linear phase hypothesis summar...	Almost done	4 months ago
README.md	Updated readme	3 months ago
config_study.txt	Refactored linear_phase_parser.py	10 hours ago

Once you have the 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 must have Python installed on the local machine.[3] After installation, the local installation directory should contain the following files and folders:



tut tiedostot > parser-grammar-master > parser-gram

- docs
- language data working directory
- lpparse
- .gitignore
- config_study
- LICENSE
- Linear phase hypothesis summary
- README

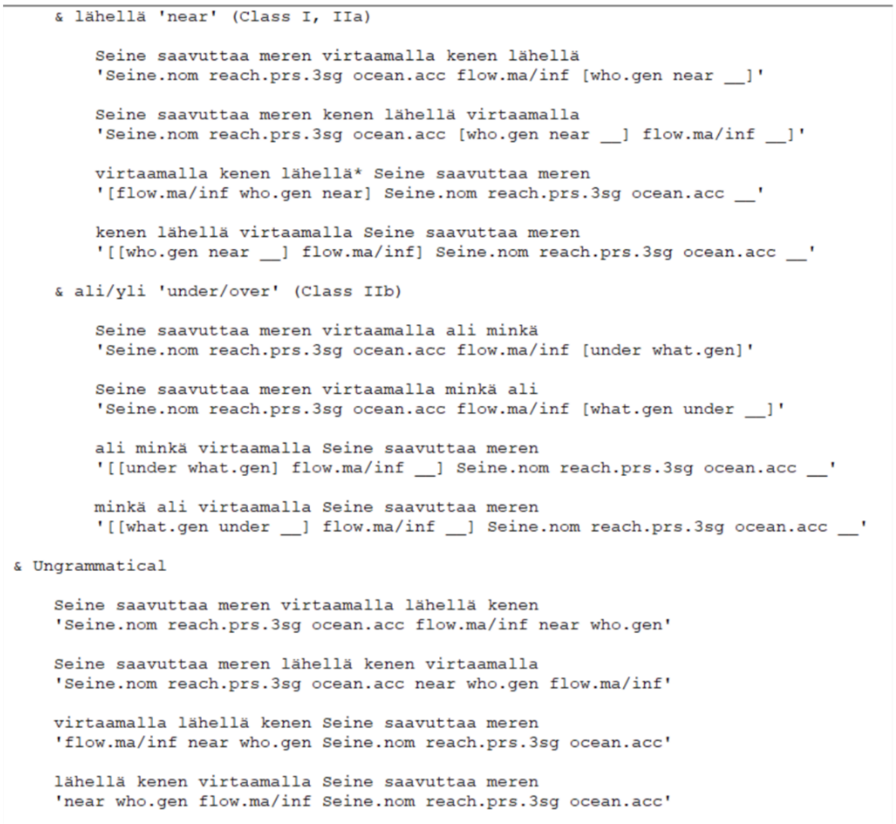
Folder */docs* contains documentation (e.g., this document and previous versions), */language data working directory* stores the input and output files associated with any particular study, and */lpparse* stores the modules containing the program code. The file *config_study.txt* is a text file that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

2.2 Use

The script is launched by having the Python interpreter to run the program script. In most cases the version copied or cloned from the code repository comes in a working configuration, it should do something meaningful out of the box.[4] To execute the script in Windows, start a command prompt program such as Windows PowerShell or the command prompt ("cmd") and navigate to the local installation directory. The script can be executed by command *python lpparse*. The program reads a study configuration file *config_study.txt* from the local installation directory containing the parameters used in the simulation trial. One of these parameters is the folder and name of the *test corpus file* that contains the sentences the script will analyze. See below.

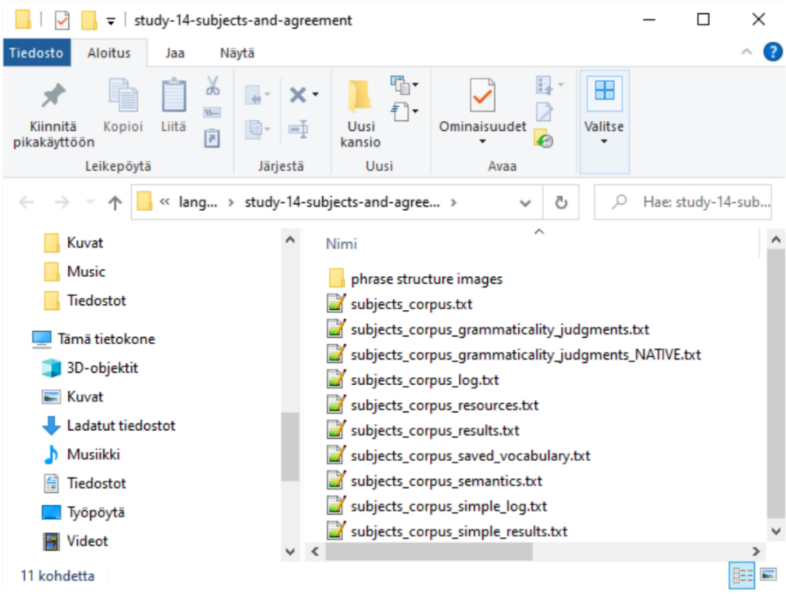
[3] Follow the instructions from www.python.org. In Windows, Python installation path must be provided in the Windows PATH environmental variable. Search for the term "setting windows PATH variables." Some versions rely on external Python libraries which provide auxiliary functionality, such as phrase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

[4] The version downloaded by following the instructions above is always the latest. If the user wants to download older versions, say for replication purposes, then these can be downloaded by first selecting the branch and then performing the instructions above; see below.



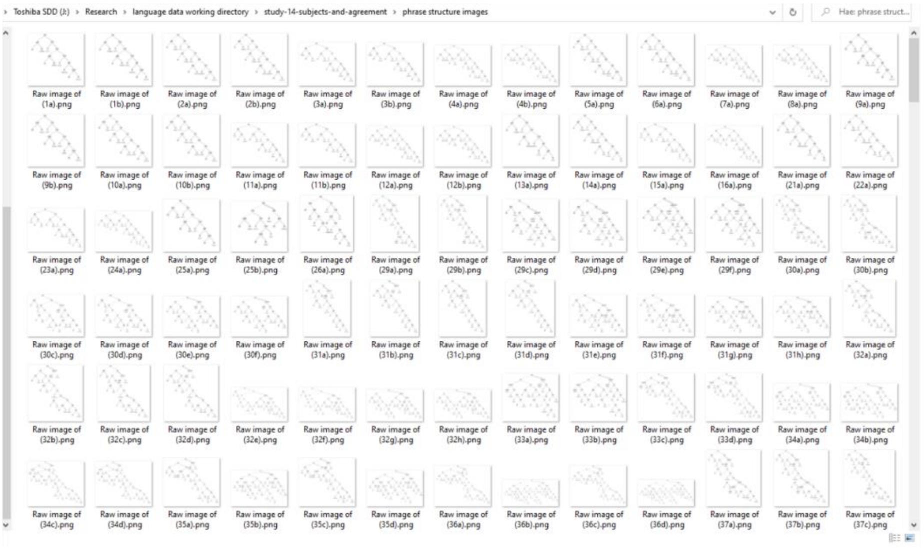
```
    & lähellä 'near' (Class I, IIa)

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

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

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

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

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

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

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

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

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

& Ungrammatical

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

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

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

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

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



phrase structure images
subjects_corpus.txt
subjects_corpus_grammaticality_judgments.txt
subjects_corpus_grammaticality_judgments_NATIVE.txt
subjects_corpus_log.txt
subjects_corpus_resources.txt
subjects_corpus_results.txt
subjects_corpus_saved_vocabulary.txt
subjects_corpus_semantics.txt
subjects_corpus_simple_log.txt
subjects_corpus_simple_results.txt

The first file is the test corpus, followed by several outputs generated by the algorithm. For example, the file *subjects_corpus_grammaticality_judgments.txt* contains a list of the original sentences and the grammaticality judgments provided by the model. These files can be opened by any text editor. The folder */phrase structure images* contains a phrase structure image for each grammatical input sentence, as calculated by the model[5]:



2.3 Structure of the script

When the user runs the script by typing *python lpparse* inside the installation director, the Python interpreter will run a main script (called *__main__.py*) from the folder */lpparse*. This folder contains the modules defining the theory and the behavior of the whole system. The *__main__.py* module will call the program code that exists inside the individual files. These files (also called modules) correspond closely to the contents of the empirical theory that they implement. For example, the module *SEM_narrow_semantics.py* contains operations that correspond to an empirical component carrying the same name in the theory. These files can be opened and edited by any text editor. Thus, if the researcher wants to find out how some grammatical operation, principle or module operates, this information can be found from these files. If the user changes any of these files and runs the script anew, then the modified script will be run, and the original output files will be overwritten. Python is an interpreted language, meaning that there is no separate executable; each time the user runs the script these text files are consulted.[6]

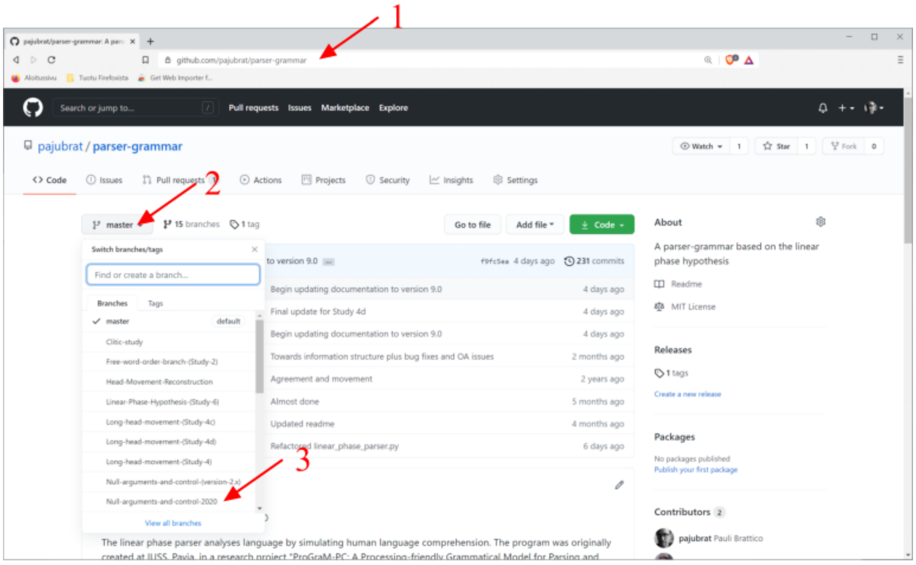
2.4 Replication

When a study is published, the input files plus its source code is stored in source code repository. The source code repository (that runs on a program called Git) not only stores the current version

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

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

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



Branches:
- ✓ master (default)
- Clitic-study
- Free-word-order-branch-(Study-2)
- Head-Movement-Reconstruction
- Linear-Phase-Hypothesis-(Study-6)
- Long-head-movement-(Study-4c)
- Long-head-movement-(Study-4d)
- Long-head-movement-(Study-4)
- Null-arguments-and-control-(version-2.x)
- Null-arguments-and-control-2020

to version 9.0 f9fc5ea 4 days ago	231 commits
Begin updating documentation to version 9.0	4 days ago
Final update for Study 4d	4 days ago
Begin updating documentation to version 9.0	4 days ago
Towards information structure plus bug fixes and OA issues	2 months ago
Agreement and movement	2 years ago
Almost done	5 months ago
Updated readme	4 months ago
Refactored linear_phase_parser.py	6 days ago

A parser-grammar based on the linear phase hypothesis

Readme

MIT License

The linear phase parser analyses language by simulating human language comprehension. The program was originally created at IUSS, Pavia, in a research project "ProGraM-PC: A Processing-friendly Grammatical Model for Parsing and

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.

2.5 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 set of default datasets were created which contain the core sections from previous datasets. These datasets are located in directory */language data working directory/standard datasets*.

3 Framework

3.1 The framework

A native speaker can interpret sentences in his or her language by various means, for example, by reading sentences printed on a paper. Since it is possible accomplish this task without external information, all information required to interpret a sentence in one's native language must be present in the sensory input. We assume that efficient and successful language comprehension is possible from contextless and unannotated sensory input.[7]

Some sensory inputs map into meanings that are hard or impossible to construct (e.g., *democracy sleeps furiously*), while others are judged as ungrammatical. A useful theory of language comprehension must appreciate these properties. The parser therefore defines a characteristic function from sensory objects into a binary (in some cases graded) classification of its input in terms of grammaticality or some related notion, such as semanticality, marginality or acceptability. These categorizations are studied by eliciting responses from native speakers. Any language comprehension model that captures this mapping correctly is said to be *observationally adequate*. Any scientific theory must be, minimally, observationally adequate, and the fact that it is observationally adequate must be shown by deductive calculation.

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

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

part. A theory of language comprehension that captures the fixed and variable components in a correct or at least realistic way without contradicting anything known about the process of language acquisition is said to reach *explanatory adequacy*.

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

Suppose we have constructed a theory of language comprehension that can be argued to be observationally adequate and explanatory. Does it also agree with data obtained from neuro- and psycholinguistic experimentation? Realistic language comprehension involves several features that an observationally adequate explanatory theory need not capture. One such property concerns automatization. Systematic use of language in everyday communication allows the human brain to automatize recognition and processing of linguistic stimuli in a way that an observationally adequate explanatory model might or might not be concerned with. Furthermore, real language processing is sensitive to top-down and contextual effects that can be ignored when constructing a theory of language comprehension. That being said, the amount of computational resources consumed by the model should be related in some meaningful way to reality. If, for example, the model engages in astronomical garden pathing when no native speaker exhibits such inefficiencies, the model can be said to be insufficient in its ability to mimic real language comprehension. We say that if the model's computational efficiency and performance behavior is in line with evidence from real speakers, it is *psycholinguistically adequate*. I will adopt this criterion as well. Psycholinguistic adequacy cannot be addressed realistically without addressing observational and explanatory adequacy, but the vice versa is not true.

Language comprehension is viewed in this study as cognitive information processing 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 put into unambiguous linear order.

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

The ultimate nature of semantic representations is a controversial issue. One way around this problem, adopted here, is to focus on selected aspects of semantic interpretation and try to predict them on the basis of the input. For example, we could decide to focus on the interpretation of thematic roles and require that the language comprehension model provides for each input sentence a list of outputs which determine which constituents appearing in the input sentence has which thematic roles. For example, we could require that the model provides that *John admires Mary* is processed so that John is judged the agent, Mary the patient, and not the other way around. The advantage is that we can predict semantic intuitions in a selective way without taking a strong stance towards the ultimate nature and implementation of the semantic notions. Another advantage is that we can focus on selected semantic properties without trying to understand how the semantic system works as a whole.

A third advantage of this approach is that we do not need to decide a priori what type of linguistic structures will be used in making these semantic predictions. We can leave that matter open for each theory to settle and require that correct semantic intuitions, in whichever way they are ultimately represented in the human brain, be predicted. Of course, any given model must ultimately specify how these predictions are generated. I will adopt what I consider to be the standard assumption in the present day generative theory and assume that input sentences are interpreted semantically on the basis of representations at a *syntax-semantics interface*. The syntax-semantic interface is considered a level of representation (ultimately a collection of neuronal connections) at which all phonological, morphological and syntactic information processing has been completed and semantic processing takes over. It is also called Logical Form or *LF interface*. I will use both terms in this document. This means, then, that the language comprehension model will map each input sentence into a set of LF interface objects, which are interpreted semantically.[8]

It is not possible to construct a model of language comprehension without assuming that there occurs a point at which something that is processed from the sensory input is used to construct a semantic interpretation. A syntax-semantic interface seems to be a priori necessary on such grounds. Different theories make different claims regarding where they position that interface and what properties it has. It is possible to assume that the interface is located relatively close to the surface and operates with linguistic representations that have been generated from the sensory objects themselves by applying only a few operations. Meaning would then be read off from relatively shallow representations. An alternative, a "deep" theory of the syntax-semantic interface, is a theory in which the sensory input is subjected to considerable amount of processing before anything reaches this stage. We can build the model by assuming that there is only one syntax-semantic interface, or several, or that the linguistic information flows into that system all at once, or in several independent or semi-independent packages. The only way to compare these alternatives, and many others that imaginable, is to examine to what extent they will generate correct semantic interpretations for a set of input sentences; it is pointless to try to use any other justification or argument either in favor or against any specific model or assumption. In general, then, we do not posit further restrictions or properties to the syntax-semantic interface apart from its existence.

3.2 Computational linguistics methodology

Scientific hypotheses must be justified by deducing the observed facts from the proposed hypothesis. The method is routinely used all advanced sciences. To do this, we begin by narrowing down a dataset based on the interests of the particular study. In linguistics, a typical dataset contains grammatical and ungrammatical expressions paired with their meanings and other attributes, but it could involve actual use patterns, communicative intuitions, or pragmatic presuppositions. This dataset is 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

[8] Almost any model can be interpreted in these terms. If we assumed that the input is something less than a linear string of phonological words, then the model must include lower-level information processing mechanisms that can preprocess such stimuli, perhaps ultimately the acoustic sounds, into a form that can be used to activate lexical items, in a specific order. We can also assume more, for example, by providing the model additional information concerning the input words, such as morphological decompositions, morphosyntactic structure or part-of-speech (POS) annotations.

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).[9] 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.

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

3.3 Overview of the hypothesis

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

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

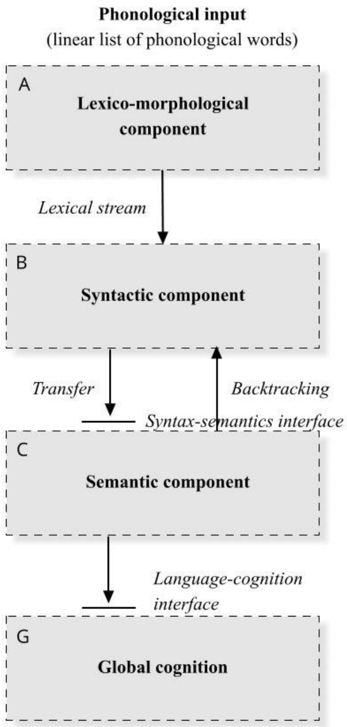

Phonological input
(linear list of phonological words)
A
Lexico-morphological component
Lexical stream
B
Syntactic component
Transfer
Backtracking
Syntax-semantics interface
C
Semantic component
Language-cognition interface
G
Global cognition

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 the incoming lexical items incrementally into a partial phrase structure representation in the current syntactic memory that has been assembled on the basis of the words seen so far. This process is illustrated in Figure 2.

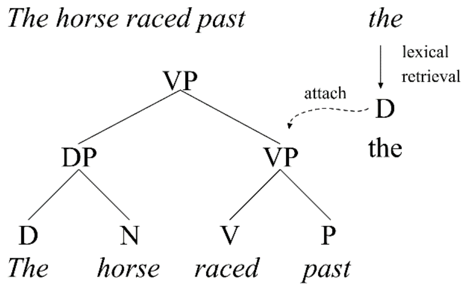


Figure 2. Operation of the syntactic component

The resulting phrase structure representation is called *spellout structure* since it corresponds transparently to the linear order in the sensory input, being directly generated form it.[10] Once all words have been consumed from the input, the result will be *transferred* to the *syntax-semantic interface* (C) where the candidate representation is evaluated. If the syntactic interpretation is grammatical and interpretable, the solution will be *accepted* and the input string will be judged grammatical. An accepted structure is forwarded to a semantic component (called narrow semantics) which provides it with detailed semantic interpretation and interacts with global cognitive processes (thinking, decision making, discourse). This corresponds to a process in which the hearer understands the input sentence within some communicative context. If the candidate solution is not grammatical and/or cannot be interpreted semantically, it is *rejected* and no semantic interpretation results. 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. If no acceptable solution emerges, the sentence is judged ungrammatical.

Suppose the input string is *the horse raced past the barn* and the syntactic module provides it with the syntactic representation [$_{\text{VP}}$[$_{\text{DP}}$ *the horse*] [$_{\text{VP}}$ *raced* [$_{\text{PP}}$ *past* [$_{\text{DP}}$ *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) [$_{\text{VP}}$ [$_{\text{DP}}$ the horse] [$_{\text{VP}}$ raced [$_{\text{PP}}$ past [$_{\text{DP}}$ 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) [$_{\text{VP}}$ [$_{\text{DP}}$ the horse [$_{\text{VP}}$ (that) raced past the barn]] fell]

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

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

When the syntactic component assembles a solution for the input, the solution will be transferred to the syntax-semantics interface for evaluation and interpretation (see the arrow "Transfer" in Figure 1). While linguistic expressions are language-specific, the system that interprets them is universal. The cognitive capacities of speakers are virtually the same independent of the language(s) they happen to speak. Transfer removes language specific properties from the input so that it can be interpreted and processed further. It detects elements that occur in "wrong" positions where they cannot be interpreted and tries to *reconstruct* them into positions in which they can be interpreted. In Finnish, for example, speakers can reverse the order of the subject and object and produce an inverted OVS sentence that is noncanonical but still grammatical (Finnish is a canonical SVO language). Transfer will reconstruct the object and the subject into their canonical positions where they can be associated at the syntax-semantics interface with universal semantic notions such as agent and patient. In this case, the reconstruction is based on the overt morphological case features of the input words (nominative = subject, partitive = direct object). The process is illustrated in Figure 3, in a highly simplified form.[11]

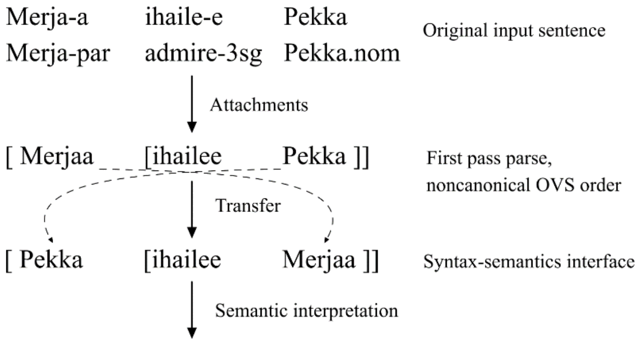

Merja-a ihaile-e Pekka
Merja-par admire-3sg Pekka.nom
Original input sentence
Attachments
[Merjaa [ihailee Pekka]]
First pass parse, noncanonical OVS order
Transfer
[Pekka [ihailee Merjaa]]
Syntax-semantics interface
Semantic interpretation

Figure 3. Transfer as an error correction mechanism.

Transfer is a cognitive reflex that is applied to all linguistic representations that are send to the syntax-semantics interface for interpretation. We can imagine it as a noise tolerance mechanism performing limited amount of reconstruction when the linguistic elements (words and their pars) appear at noncanonical positions.

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

At any given moment during a linguistic conversation or communication the hearer maintains a transitory repository of semantic objects called *global discourse inventory* that the conversation "is about." Thus, if we talk about a person called John, the discourse inventory will contain a representation of John. The objects maintained in the discourse inventory are language-external 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 viewed as a process in which the output of the syntactic pathway is converted into changes in the language-external discourse inventory. Each sentence adds, removes or updates elements and their properties in this repository. This conversion happens inside *narrow semantics*. We can therefore conceptualize narrow semantics as a structure that mediates communication between the language faculty, the syntactic pathway more specifically, and the language-external systems that constitute global cognition and access the discourse inventory of transient semantic objects activated during the conversation or communication. A *conversation* is defined as a sequence of sentences that share the global discourse inventory, making it possible to use introduction sentences for populating the inventory and test sentences that make claims about those entities (e.g., *John*$_1$ *admires Mary*$_2$; *he*$_1$ *likes her*$_2$).

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 the published literature and are omitted here.

4.2 Merge-1

Linguistic input is received by the hearer in the form of sensory stimulus. We can first think of the input, to simplify the situation to bare essentials, as a one-dimensional string α * β * … * γ of phonological words. In order to understand what the sentence means, the human parser (which is part of the human language faculty) must create a sequence of abstract syntactic interpretations for the input string received through the sensory systems. One fundamental concern is to recover the hierarchical relations between words. Let us assume that while the words are consumed from the input, the core recursive operation in language, Merge, arranges them into a hierarchical representation. For example, if the input consists of two words α * β, Merge yields [α, β](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 they are represented as being part of the same linguistic chunk. We can attend to both of them as one object, manipulate them as part of the same representation, and in general perform operations that takes them both into account. Therefore, operation (3) presupposes that there exists a formally defined notion of phrase structure that is able to represent entities of this type.

Example (3) suggests that the syntactic component combines phonological words. While this is a possibility, it would not be linguistically useful assumption. We assume that (3) is mediated by *lexicon*: a storage of linguistic information that is activated on the basis of the original phonological words. The lexicon maps phonological words in the input into *lexical items* which contain *lexical features* such as lexical categories (noun, verb), inflectional features ('third person singular') and meaning ('John', 'sleeping'). 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 with phonological words.[12]

The resulting complex chunk [α, β] is asymmetric and has a *left constituent* and a *right constituent*. The terms "left" and "right" are mnemonic labels and do not refer to concrete leftness or rightness at the level of neuronal implementation. Their purpose is to distinguish the two constituents from each other. They are related to leftness indirectly: since we 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*. Example (4) shows three possible attachment sites, all which correspond to different hierarchical relations between the words.

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

The operation illustrated in (4) differs in a number of ways from what constitutes the standard theory of Merge at the time of present writing. I will label the operation in (4) by Merge-1, symbol -1 referring to the fact that we look the operation from an inverse perspective. Instead of generating linear sequences of words by applying Merge, we apply Merge-1 on the basis of a linear sequence of words in the sensory input and assemble the structure from left to right. The ultimate syntactic interpretation of the whole sentence is generated as a sequence of partial phrase structure representations, first [*John*], then [*John, sleeps*], then [*John* [*sleeps, furiously*]], and so on, until all words have been consumed from the input.

Several factors regulate Merge-1. One concern is that the operation may in principle create a representation that is ungrammatical and/or uninterpretable. Alternative (4)(a) can be ruled out on such grounds: *John furiously* is not an interpretable fragment. Another problem of this alternative is that it is not clear how the adverbial, if it were originally merged inside subject,

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

could have ended up as the last word in the linear input. The default left-to-right depth-first linearization algorithm would produce **John furiously sleeps* from (4)a. Therefore, this alternative can be rejected on the grounds that the result is ungrammatical and not consistent with the word order discovered from the input. If the default linearization algorithm proceeds recursively in a top-down left-right order, then each word must be merged to the *right edge* of the phrase structure, right edge referring to the top node and any of its right daughter node, granddaughter node, recursively. Under these assumptions a left-to-right model will translate into a grammatical model in which the grammatical structure is expanded at the right edge.

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

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

The parser will merge *furiously* into a right-adverbial position (b) and branch off recursively. If this solution does not produce a legitimate output, it will return to the same point and try solution (c). Every decision made during the parsing process is treated in the same way. All solutions constitute potential phrase structures, which are ordered in terms of the ranking. The mechanism can be illustrated with the help of a standard garden path sentence such as (6).

(6)
a. The horse raced past the barn.
b. The horse raced past the barn fell.

Reading (6)b involves extra effort when compared to (a). At the point where the incremental parser encounters the last for *fell*, it has (under normal language use context) created a partial representation for the input in which *raced* is interpreted as the past tense finite verb, hence the assumed structure is [$_{\text{DP/S}}$ *The horse*][$_{\text{V}}$ *raced*][$_{\text{PP}}$ *past the barn*] (DP/S = a subject argument, V = verb, PP = preposition phrase). Given this structure, there is no legitimate right edge position into which *fell* could be merged. Once all these solutions have been found impossible, the parser backtracks and considers if the situation could be improved by merging-1 *barn* into a different position, and so on, until it discovers a solution in which *raced* is interpreted as a noun-internal relative construction [$_{\text{DP}}$ *The horse raced past the barn*] *fell*. This explanation presupposes that all solutions at each stage are ranked, so that they can be explored recursively in a well-defined order. Thus, at the stage at which raced is merged-1, the two solutions *raced* = finite verb and *raced* = participle are ranked so that the former is tried first.

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

(7) John sleeps furiously
↓ ↓ ↓
[John, sleeps] → [John [sleeps furiously]]

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

(8) [John [[sleeps furiously] γ]

One consequence of this is that if we merge two words as sisters, we do not know if they will maintain the same or any close structural relationship in the derivation's future. In (8), they don't: future merge operations break up constituency relations established earlier. Consider the stage at which *John* is merged with a wrong verb form *sleep*. The result is a locally ungrammatical string **John sleep*. But because constituency relations can change in the derivation's future, we cannot rule this step locally as ungrammatical. It is possible that the verb 'sleep' ends up in a structural position in which it bears no meaningful linguistic relation with *John*. Only those configurations or phrase structure fragments can be checked for ungrammaticality that cannot be tampered in the derivation's future. Such fragments are called *phases* in the current linguistic theory.

4.3 Lexical selection features

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

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

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

(10) John admires Mary
↓ ↓ ↓
[John [admires Mary]]
[!COMP:D] [D]

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

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”) favors solution (a), [SPEC:V] favors solution (b)(“select the verb phrase”). What constitutes a specifier selection feature will guide the selection of a possible left sister during comprehension.[13] In sum, lexical features are used for guiding the parsing process towards meaningful solutions and later for checking that any given full solution is grammatical and/or can be interpreted.

4.4 Phases and left branches

Let’s consider the derivation of (12).

(12) John’s mother admires Mary.
↓ ↓ ↓ ↓
[$_S$[$_{DP}$ John’s mother] [$_{VP}$ admires Mary]]

After the finite verb has been merged with the DP *John’s mother*, no future operation can affect the internal structure of that DP. Merge-1 is always to the right edge. All left branches therefore become *phases*: the derivation can forget them inside that particular parsing path. 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 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. Since the model can backtrack, it is able to reconsider all left branches.

4.5 Labeling

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

[13] Because constituency relations may change later, we do not know which pairs will constitute specifier-head relations in the final output. Therefore, we use specifier selection to guide the parser in selecting left sisters, and later use them at the syntax-semantics interface (LF interface) to verify that the output contains proper specifier-head relations.

(14) Mary   admires   John's   mother
  ↓   ↓   ↓   ↓
Mary   [admires [John's   mother]]

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? Since any constituent may contain an arbitrary number of elements, there is no trivial or self-evident mapping from constituents into labels like [D]. The mapping from constituents into labels 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. Closest means closest form the point of view of the selector. If we analyzed the situation from the point of view of the labeled phrase itself, then closest is the highest or most dominant head. Principle (15) means that labeling ignores right adjuncts (this will be a defining feature of adjuncts). Example (16) shows some basic examples. Notice how the labels of complex phrases are written inside the brackets, thus [$_{\text{VP}}$ …] means a verb phrase whose label [V] has been calculated by (15).

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

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

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

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

e. [$_{\text{VP}}$[$_{\text{VP}}$ *admires Mary*]〈*furisously*〉] = adjunct is ignored;

f. [$_{\text{NP}}$ [$_{\text{DP}}$ *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 derivation:

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

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

What was originally a DP was transformed into a VP after the verb was merged. We do not need to fit phrase structure templates to the input; rather, the input generates an appropriate phrase structure.

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.
↓ ↓
[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*]. However, (18) is a sentence or verb phrase, not a DP. We assume that *John* is a complex constituent despite of its appearance and has structure is [$_{\text{DP}}$ D N]. This information comes from the lexicon, where proper names are decomposed into D + N structures. The structure of (18) is therefore (19).

(19) John + sleeps.
↓ ↓
D N sleeps
↓ ↓ ↓
[$_{\text{VP}}$[$_{\text{DP}}$D N] sleeps]

4.6 Upward paths and memory scanning

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

(20) a. Pekka ei ihaillut *Merja-n/ Merja-a.
Pekka not admire Merja-acc Merja-par
’Pekka did not admire Merja.’
b. Pekka on ihaillut Merja-n/ *Merja-a.
Pekka has admired Merja-acc Merja-par
’Pekka did admire Merja.’

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

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

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

(22) *Upward path*
the upward path from α contains all constituents that dominate α plus their immediate daughters.

For example, if the phrase structure is (23)

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

then the upward path from α contains αP, BP and AP (the dominating constituents) plus their immediate constituents A and B. The dependency itself is formed by scanning through the path for a target element, and the first target element available is always selected. If the target is not found, the checking operation fails. This mechanism is applied to case checking, binding, adjunct positioning, control and operator scope calculations among many other nonlocal grammatical operations. Intuitively the path defines or coincides with the content of the “syntactic working memory” when α is processed.[14]

4.7 Adjunct attachment

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

[14] Recall that all left branches are phases (Section (13)), thus they have already been processed and moved out of the active syntactic working memory when α is targeted for operations.

(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 Mejaa *ilmeisesti*
Pekka admires Merja apparently

The adverbial *ilmeisesti* ‘apparently’ can occur almost in any position in the clause. Consequently, it will be merged into different positions in each sentence. This confuses labeling and present a challenge for the parser, because the position of the adverb is almost completely unpredictable. 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. They increase the dimensionality of the phrase structure. This hypothesis is illustrated in Figure 24.

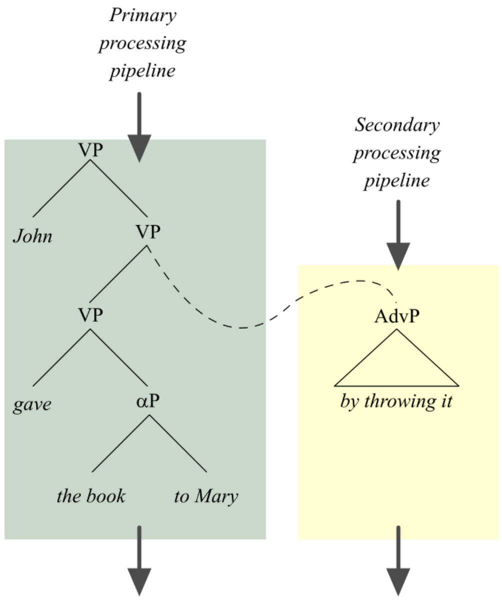


Figure 24. Adjuncts are geometrical constituents that are pulled out of the primary working memory and are processed inside separate processing pipeline. We can imagine that the VP is made up of two segments.

Thus, the labeling algorithm as specified in 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) [$_{\text{VP}}$ John [$_{\text{VP}}$[$_{\text{VP}}$ admires Mary] 〈$_{\text{AdvP}}$ furiously〉]]

Consider (26) next.

(26) John [sleeps 〈$_{\text{AdvP}}$ 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 having 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.[15] The fact that adjuncts, like the adverbial *furiously* here, are optional follows from the fact that they are automatically excluded from selection and labelling. Whether they are present or absent has no consequences for either of these dependencies.

It follows from these assumptions, however, that adjuncts could be merged anywhere, which is not correct. Each adverbial (head) is associated with a feature linking it with a feature or set of features in its hosting, primary structure. In this case the linking relation is established by a *tail-head relation*: we imagine that the adjunct, which exists in the separate syntactic plane or pipeline, constitutes a hanging tail related to the head. For example, a VP-adverbial is linked with V, a TP-adverbial is linked with T, a CP-adverbial is linked with C. Linking is established by checking that the adverbial (i) occurs inside the corresponding projection (e.g., VP, TP, CP) or 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.[16]

(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).

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

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

B_F denotes a head (or phrase) that has feature [F]. Condition (27)(a) is relatively uncontroversial. It states that a VP-adjunct must occur inside VP, or more generally, αP adjunct has to be inside a projection from [α], where α is a feature. It does not restrict the position at which an adjunct must occur inside αP. Condition (27)(b) allows some adjuncts, such as preposition phrases, remain in a low right-adjoined or in "extraposed" positions in a canonical structure. If an adverbial/head does not satisfy a tail feature, it will be reconstructed into a position in which it does during transfer. This operation will be discussed in 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 removing most or in some cases all language-specific properties from the input and deliver it in a format understood by the universal semantic system and the universal conceptual-intentional systems responsible for thinking, problem solving and other language-external cognitive operations that are part of the global cognition.

4.8.2 Phrasal reconstruction: general comments

A phrase or word can occur in a *canonical* or *noncanonical* position. A canonical position in the input could be defined as a position such that given the regular Merge-1 operations a constituent will 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]]

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

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

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

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

This solution violates two selection rules. There are two specifiers at the left edge, *who* and *Pekka*, and *admire* lacks a complement. Furthermore, even if the parser backtracks, it cannot find a legitimate solution. A candidate structure such as [$_{\text{VP}}$ [$_{\text{DP}}$ *who Pekka*] *admires*] is also illegitimate.[17] The two violations are not independent of each other: the element that triggers the double specifier violation at the left edge is the same element that is missing at the complement position. This is clearly not a coincidence: the interrogative pronoun causes these problems because it has been "dislocated" to the noncanonical position from its canonical complement position.

Dislocation is a persistent and general feature of all natural languages. In Finnish, for example, almost all word order variations of the referential arguments of a finite clause are possible. 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. For example, the two orders in (31) correspond to the same core proposition 'Pekka admires Merja'.

(31) a. Pekka ihailee Merja-a. (SVO)
Pekka.nom admires Merja-par
'Pekka admires Merja.'
b. Merja-a ihailee Pekka. (OVS)
Merja-par admires Pekka.nom
'Pekka admires Merja.'

Unless the system perform normalization where all selection restrictions can be checked, we cannot filter out ungrammatical sentences from the grammatical ones. It is clear, furthermore, that Finnish speakers use overt case forms, here the nominative and partitive, to do this. The model

[17] In theory we could feed (30) directly to the LF interface component. This representation does not satisfy the complement selection features of the verb *admire*. If we do not control for what is filling the complement position of the verb, the model would accept **John admires* or even **John admires Mary to leave*. Similarly, the main verb is here preceded by two argument DPs, but this too can produce ungrammatical results such as **John Mary admires*.

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

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

4.8.3 Ā-chains

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

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

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

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

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

The search algorithm proceeds from the deviant element downstream until either a suitable position is found or there is no more structure. It cannot go upwards or sidewards. In this case the operation begins from the sister of the targeted element and proceeds downward while ignoring left branches (phases) and all right adjuncts. This operation is called *minimal search*. 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 was always interpreted (implicitly, as it were) as a declarative proposition. Interrogativization is represented by a *wh*-feature that must be part of the highest head in the sentence, so that it can be selected by *ask*/*claim*. This is accomplished by copying the *wh*-feature from the fronted interrogative pronoun to the local head (34)(a) or, if no local head is available, it is generated to the structure and then equipped with the *wh*-feature (b).[18]

(34) a. Who admires Pekka?
$[_{VP}$ who$_{wh}$ $[_{VP}$ admires$_{wh}$ Pekka]]
b. who Pekka admires?
~~who$_{wh}$~~ C$_{wh}$ [Pekka [admires who$_{wh}$]]

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

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

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

The dislocated phrase also represents the *propositional scope* of the question. The interpretation is one in which the speaker is asking for the identity of the object of admiration (‘which x: Pekka admires x’). Let us consider how dislocation represents the propositional scope of an interrogative clause. First, the reconstructed interrogative pronoun or phrase is called an *operator*. This means

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

that an argument such as ‘who’ does not refer to anything by itself; it is an “argument placeholder.” We can also think of it as a variable, the target of the question that must be filled in by the “answer.” Then, any finite element that contains the same operator feature, in this case *wh*-feature, will determine the scope.

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

The operator-scope dependency, marked by the line in the representation above, is checked by an operator-variable module inside narrow semantics, which pairs the operator with its scope marker. If feature [OP] (e.g. *wh*) represents the operator, then the closest head with [FIN][OP] will be the scope marker. These dependencies, which have both a syntactic and semantic dimension, are called *Ā-dependencies* or *Ā-chains* in the literature. Chains have a syntactic side, because the dislocated deviant element must be reconstructed by a syntactic operation that takes place during transfer, by minimal search; and a semantic side, because the operator must be linked with the corresponding scope marked inside the semantic system. Both operations can fail, and when one or both fail, the input sentence is judged ungrammatical.[19]

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.

4.8.4 A-chains and EPP

In addition to Ā-chains, discussed in the previous section, many languages exhibit another type of phrasal reconstruction, called A-chains. English 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 [was [admired (by John)]]]

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

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

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

Tense T can be expressed either by an auxiliary (*was*, *does*) or by combining it with the main verb; the latter operation will be discussed in a subsequent section. 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: 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. 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. We will examine the EPP issue in Section 4.9. A-reconstruction differs from Ā-reconstruction in that the former does not form operator-variable constructions inside the semantic system and is not triggered by operator features such as *wh*-features.[20]

4.8.5 Head reconstruction

Let us consider again (40).

[20] The EPP no longer exists as a monolithic lexical feature but is replaced by a function EF that relies on a set of edge features. When these details do not matter I will keep using the term "EPP." See Section 4.9.

(40) John admires Mary.

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

First we should note that whether and how grammatical heads are chunked into morphological and phonological words is to some extent arbitrary and subject to crosslinguistic variation. Thus in English, we can express the same tense information by *John does admire Mary* and *John admires Mary*, where in the latter the two heads have been chunked together. Head reconstruction must therefore contain an operation that recovers morphological chunks from phonological words. This part is handled by the lexico-morphological module. It takes phonological words as input and decomposes them into their constituent morphemes, thus /admires/ ~ T_{prs} + V. These heads are forwarded into the syntactic component and merged-1 to the syntactic structure. A *complex head* $T(V)^0$ is created inside the syntactic component. Thus, the lexical streaming operation maps lexical decompositions into complex heads: T + V ~ $T(V)^0$. Head reconstruction extracts the heads and distributes them into the structure, $T(V)^0$ ~ [T … [...V…]]. The steps are illustrated in (41).

(41) John admires Mary. (Input)
[John [$T(V)^0$ Mary]] (Output of Merge-1)
[John [T [V Mary]]] (Output of head reconstruction)

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

We assumed earlier that the phrase structure is made up of primitive constituents, which are elements that have zero daughters, and complex constituents, which have two daughters. Thus, if α and β are primitive constituents, then [α β] is a legitimate complex constituent. Because [α β] 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 $\alpha(\beta)^0 = [_\alpha\ \beta]$, where α is a constituent that has only one (right) constituent β. The mechanism is iterative, and allows the system to chain several heads into ever more complicated words, for example $A(B(C)))^0 = [_A [_B C]]$. Phrasal syntactic operations do not treat nonphrasal complex constituents (complex heads) as phrases (observing the lexical integrity principle), while they can still contain several grammatical heads (primitive lexical items) ordered into a linear sequence.[21]

4.8.6 Adjunct reconstruction

Thus far we have examined Ā-reconstruction, A-reconstruction and head reconstruction. There is a fourth reconstruction type called *adjunct reconstruction* (also adjunct floating, scrambling). Consider the pair of expressions in (42) and their canonical derivations.

(42)

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

Derivation (b) is incorrect. Native speakers interpreted the thematic roles as being identical in these examples. The subject and object are in wrong positions. Neither Ā- nor A-reconstruction can handle these cases. The problem is created by the grammatical subject *Pekka* ‘Pekka.nom’, which has to move upwards/leftward in order to reach the canonical LF-position SpecVP. Because the distribution of thematic arguments is very similar to the distribution of adverbials in Finnish, I have argued that richly case marked thematic arguments can be promoted into adjuncts. Case forms, according to this view, are 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 (43). Notice that because the grammatical subject *Pekka* is promoted into adjunct, it no longer constitutes the complement; the partitive-marked direct object does.

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

(43) [⟨Merjaa⟩$_2$ T/fin [__$_1$ [käski [meidän [ihailla [__$_2$ ⟨Pekka⟩$_1$]]]]
Merja.par asked we.gen to.admire Pekka
'Pekka asked us to admire Merja.'

4.8.7 Minimal search

Let's consider again a standard interrogative sentences such as (44).

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

We have assumed that the interrogative pronoun is reconstructed into the canonical complement position marked by the gap __. The fronted interrogative and the gap form an Ā-chain. What has been left unsaid is how this reconstruction actually happens. It is based on *minimal search*. The gap position is located by descending downwards through the phrase structure from the position of the reconstructed element and detecting the first legitimate position (hence the term "minimal"). More formally, at each step the algorithm assumes a phrase structure γ as input and moves to α in $\gamma = \{_{\alpha P}\ \alpha\ \beta\}$ unless α is primitive, in which case β is targeted.[22] The operation therefore moves downwards on 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. Consider (44). To reach the reconstruction site __$_1$, the search algorithm must avoid going into the subject A and into the temporal adverbial B. This is prevented by minimal search, which follows the projectional spine of the sentence and ignores both left specifiers/adjuncts and right adjuncts (there are no "right specifiers" in the model).

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

4.8.8 Agree-1

Most languages exhibit an agreement phenomenon, in which some element, such as finite verb, agrees with another element, typically the grammatical subject. This is illustrated in (45).

(45) 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,

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

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 (45).

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 EF.[23] A lexical item without EF does not exhibit phi-agreement. In English, conjunctions (*but*, *and*) and the complementizer (*that*) belong to this class. Those lexical items which can exhibit agreement have feature EF. 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 ±ARG. Negative marking −ARG creates concord, the positive marking +ARG forces the element to get linked with a full argument DP. This linking will be interpreted at LF interface as *predication*. These features leave room for a predicate that is linked with an argument but does not phi-agree. This group creates control, discussed later. The four options are illustrated in Table 1.

Table 1. Four agreement signatures depending on features ±VAL and ±ARG.

	−EF (no EF)	+EF
−ARG	Lexical items exhibiting neither agreement nor require arguments (particles, such as *but, also, that, not*)	Lexical items which exhibit agreement but do not require linking with arguments (agreement by concord, e.g. *piccolo, pienet*)
+ARG	Lexical items which do not exhibit agreement but require linking with arguments (control constructions, such as *to leave, by leaving*)	Lexical items which exhibit agreement and linking with arguments (finite verbs, *admires*)

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 [φ_]. The value will be provided by the argument with which the predicate is linked with. This is shown in (46).

[23] ±VAL in earlier models. The agreement phenomenon was unified with the EPP mechanism and is therefore currently triggered by the edge feature EF which is, in turn, implemented by a separate function EF().

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

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 (47).

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

As a consequence, 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 can 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* 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 (48) comes from Italian.

(48) 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 (49).

(49) 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 (50).

(50) a. admire- = lexical predicate, hence it has unvalued phi-features [φ_];
b. -s = third person singular valued inflectional phi-features [φ].
c. = [φ_] + [φ].

Existing valued phi-features impose two constraints to the operation. First, Agree-1 must check that if the head has valued phi-features, no phi-feature conflict arises. Thus, a sentence such as **Mary admire John* will be recognized as ungrammatical. Second, we allow Agree-1 to examine the valued phi-features inside the head if (and only if) no overt phrasal argument is found. The latter mechanism will create subjectless pro-drop sentences. We thus interpret a valued phi-set inside a head as if it were a truncated pronominal element (51).

(51) 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.[24]

4.8.9 Ordering of operations

Ā/A-reconstruction and adjunct reconstruction presuppose head reconstruction, because heads and their lexical features guide Ā/A- and adjunct reconstruction. The former relies on EPP features and empty positions, whereas the latter relies on the presence of functional heads. Furthermore, Ā/A-reconstruction relies on adjunct reconstruction: empty positions cannot be recognized as such unless orphan constituents that might be hiding somewhere are first returned to their canonical positions. The whole transfer sequence is therefore (52).

(52) *Ordering of operations during transfer*
a. Reconstruct heads →

[24] The mechanism defined in the agreement module are sufficient to calculate nontrivial datasets, but the implementation looks ad hoc. At a deeper level these operations most likely have to do with semantic indexing, discussed in Section 4.11.6, but the hypothesis still remains to be implemented.

b. Reconstruct adjuncts →

c. Reconstruct A/Ā-movement →

d. Agree-1 →

e. LF-interface and legibility →

f. Semantic interpretation.

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

The sequence provided in (52) is compatible with two possible implementations, one where all operations are executed during one cycle and another where they form several cycles. In an ideal case, the algorithm would travel through the structure only once and fix all errors, in the order of (52), as they are encountered. This corresponds to the first option. This unification has not been accomplished in the present version due to the fact that the adjunct reconstruction algorithm (b) still differs too much from (a), (b) and (c). In addition, the transfer algorithm triggers separate extraposition operations that fixes certain issues with adjuncts. It is clear that the adjunct handling is not optimal and, once this issue has been sorted out, the whole sequence (52) could possibly be executed inside just one cycle.

4.9 EPP and the edge feature

Let us consider the English passive sentence (53), where we assumed that the grammatical subject is reconstructed into the complement position of the verb where it receives the thematic role of the patient.

(53) Mary was admired (by John).
[Mary [was$_{[EPP]}$ [admired (by John)]]]
[Mary$_1$ [was$_{[EPP]}$ [admired __$_1$] (by John)]]]

This analysis presupposes that (i) the grammatical subject is reconstructed from SpecTP into CompVP and that (ii) it is not reconstructed further from the CompVP position. The reason we do not want to reconstruct further is because the argument receives its correct thematic role at CompVP. The SpecTP, on contrast, is not associated with any thematic role. The distinction between thematic and nonthematic positions was captured in earlier models by relying on the lexical EPP feature, shown in (53), which triggered the reconstruction operation. The idea was that the specifier position of an EPP head was not alone sufficient to provide the element a full semantic interpretation. The problem, however, was that the postulated EPP feature formed a theoretical "island," a curios exception. In a later models a generalization was proposed which assumes that the EPP behavior regulates all edge behavior of lexical items, where the notion of edge refers to extended subjects (54)a, specifiers more generally (54)b and to agreement clusters (pro-elements) (54)c, all noncomplement elements.

(54) a. *Mary* was admired __.
b. *Who* did Mary admire __?
c. (Minä) ihaile-*n* Merja-a
(I.nom) admire-1sg Merja-par

These elements are extended in the sense that they require reconstruction in order to be interpreted fully, which means that A- (a) and Ā-chains (b) are triggered by the edge feature. Because agreement was unified with the EPP, this feature also triggers agreement, operation Agree-1 (Section 4.8.8). Therefore, the original EPP feature was generalized so that it regulates phrasal specifiers, agreement and chain formation. The edge feature is abbreviated as [EF].

4.10 Lexicon and morphology

4.10.1 From phonology to syntax

Most phonological words enter the system as polymorphemic units. The lexico-morphological component decomposes phonological words into its constituent components. 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, retrieve the corresponding primitive lexical items $M_1 + \ldots + M_n$ from the same lexicon which are fed into syntax, where they form complex heads $M_1(M_2\ldots(M_n))^0$. Head reconstruction extracts them into head spreads $[M_1\ldots[M_2\ldots[\ldots M_n\ldots]]]$. The whole process is illustrated in (55).

(55) 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)

4.10.2 Lexical items and lexical redundancy rules

Primitive lexical items are sets of features, which emerge from three distinct sources. One source is the language-specific lexicon, which stores information specific to lexical items in a particular language. For example, the Finnish sentential negation behaves like an auxiliary, agrees in φ-features, and occurs above the finite tense node in Finnish. Its properties differ from the English negation *not*. Some of these properties are so idiosyncratic that they must be part of the language-specific lexicon.

Another source of lexical features comes from lexical redundancy rules. For example, the fact that transitive verbs can select object arguments need not be listed separately in connection with each transitive verb. This pattern is provided by redundancy rules which are stored in the form of feature implications '$F_1 \ldots F_n \rightarrow G_1 \ldots G_m$' which determine that if a lexical item has features '$F_1 \ldots F_n$' it will also get features '$G_1 \ldots G_m$'. 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 specify the language that particular lexical item belongs to.

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

4.10.3 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 via head reconstruction. Inflectional features are mapped into features which are inserted inside the closest lexical item in the lexical stream. 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 ~ $T_{3sg}(V)^0$ ~ [...[T_{3sg} [...V...]]. If we specified in the lexicon that 3sg refers to a derivational element, then a separate "agreement head" Agr would be generated.

4.11 Narrow semantics

4.11.1 Syntax, semantics and cognition

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

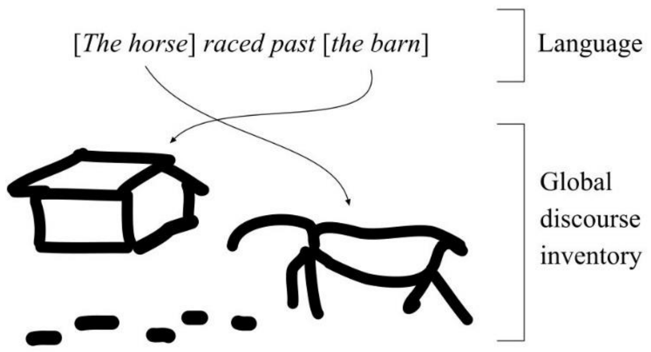


Figure 37. Narrow semantics projects semantic objects into the global discourse inventory.

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

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

Different grammatical features are interpreted by qualitatively different semantic systems. Information structure (notions such as topic and focus) is created by different processing pathway than the system that interprets quantifier scope. 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).

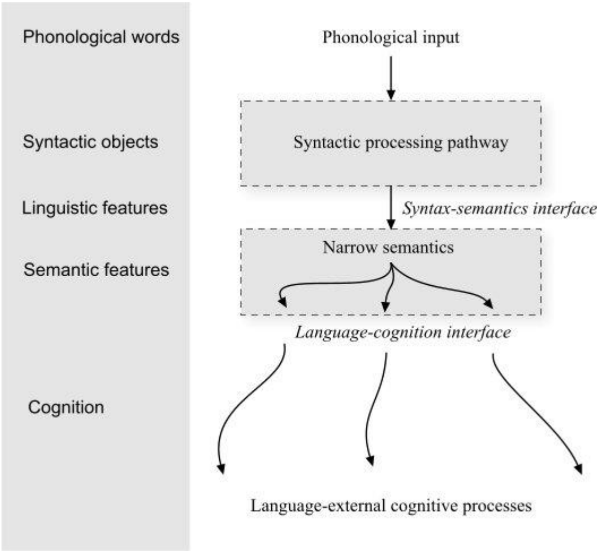

Phonological words
Syntactic objects
Linguistic features
Semantic features
Cognition
Phonological input
Syntactic processing pathway
Syntax-semantics interface
Narrow semantics
Language-cognition interface
Language-external cognitive processes

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

A proper name such as *John* can be thought of as referring to a simple “thing” like an individual person. A pronoun like *he*, on the other hand, can refer in principle to several objects in the discourse inventory (*John*$_1$ *admires Simon*$_2$*, and he*$_{1,2}$ *is very clever*), and the same is true of quantifiers like *every man* or *two men*. Furthermore, expressions like *no one* do not refer to anything, yet they, too, have meaning. Even *John* can be ambiguous if there are several men with the same name. The general problem is that there is nothing in the expression itself that determines unambiguously what it denotes, so the listener must 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.

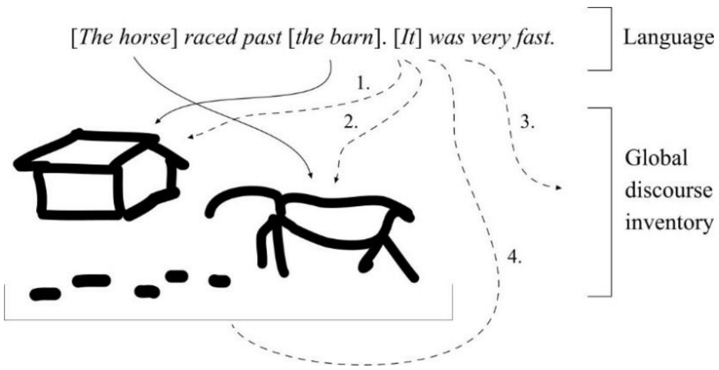


Figure 39. Possible denotations for expression *it*.

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

When a linguistic feature such as [SG] 'singular' is transformed into a formal understood by global cognition, what we mean is that the formal signal representing that feature in the linguistic output will activate a corresponding signal or representation (representing, say, 'one') inside the extralinguistic cognitive system. In the case of quantifiers such as *some*, the mapping is more complex but the principle is the same. This quantifier signals that we are supposed to select some (but no matter what and how many) objects from the global discourse inventory. I assume that the operation of 'selecting some' is part of the human cognitive repertoire accessed by narrow semantics, and that this is the reason a quantifier (or a lexical feature corresponding to it) can exist.

4.11.2 Argument structure

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

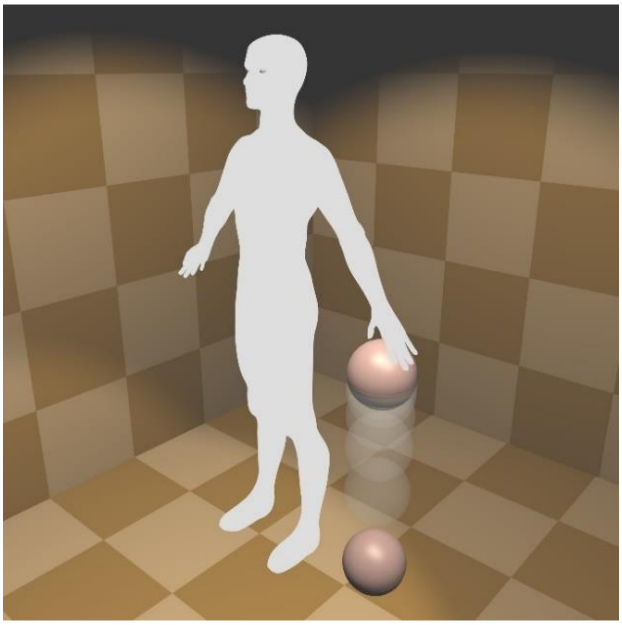


Figure 8. A non-discursive, perceptual or imagined representation of the meaning of the sentence *John dropped the ball*.

Imagine you have an non-discursive (analogous, continuous) experience of a person dropping a ball. We assume that instead of a static figure, you are looking at a dynamic scene that evolves over time. The visual-conceptual experience will be structured in such a way that we perceive the situation as involving a spatiotemporally continuous person, the ball and an event where the person drops the ball and the ball falls as a consequence. Finally, the event ends when the ball touches the floor. These are the entities that would exist, or come into existence, in the discourse inventory at the moment we experience the event. The grammatical elements that make up the sentence *John dropped the ball* grammaticalize these conceptual objects, in this case John, the ball and the event of dropping expressed by the verb. Part of the meaning emerges from the structure into which they have been embedded. Lexical items and their structure are linked with the contents of the discourse inventory and, ultimately, the everyday human experience. Let us consider the grammatical representation of the sentence (56) and how the different parts may correlate with the conceptual experience depicted in Figure 8.

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

The combination of the object the ball and the verb, the lower VP, correspond to a subevent where the ball falls. Thus, a bare verb together with an object is typically interpreted as denoting an event that involve one participant. This is combined with the small verb v and the agent argument John, which is interpreted as the causer of the subevent. That is, John is the agent of the event because he causes the ball to fall. The combination of *fall* + v will be the transitive verb *drop* ‘cause to fall’. Consequently, the argument at SpecvP is interpreted as the agent of the event, while the argument inside the VP is interpreted as the patient that must undergo the event caused by the agent. These interpretations are created when the representation arriving at the LF-interface is interpreted semantically by narrow semantics. Tense T adds tense information to the event, but the argument at SpecTP will not get a secondary thematic interpretation. In sum, the lexical items and their surrounding structures map into entities in the discourse inventory and human experience, in this case objects, events, causation and temporal properties. The elements must be organized in some specific way in order for this interpretation to succeed, and the formal selection restriction features present in the lexicon guide the syntactic processing pathway towards solutions which satisfy these ultimately semantic requirements grounded in the structure of human experience.[25]

4.11.3 Antecedents and control

Predicates are characterized by the fact that they must be linked with arguments (Section 4.8.8). The assumptions specified in the section elucidating Agree-1 leave room for a situation in which an unvalued phi-feature or a whole phi-set arrives to the LF interface unvalued. This can happen for two reasons. Recall that unvalued phi-features are normally valued by Agree-1. If the operation fails, unvalued phi-features may remain unvalued until LF-interface. Agree-1 may fail to apply if either no local argument is present (57)a-b or (2) the predicate is marked for –EF which prevents it from agreeing with anything (57)c.

(57) 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.'

An unvalued feature triggers 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

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

located by establishing an upward path (22), Section 4.6, from the triggering feature/head to the antecedent. The resulting antecedent relations are illustrated in (58).

(58) a. Pekka sanoi että __ *haluaa*$_{\varphi=\text{Pekka}}$ nukkua.
Pekka said that want-3sg sleep

b. John wants to *sleep*$_{\varphi=\text{John}}$.

c. Pekka halusi *nukku-a*$_{\varphi=\text{Pekka}}$.
Pekka wanted sleep-A/INF
'Pekka wanted to sleep.'

If LF-recovery finds no antecedent, the argument is interpreted as generic, corresponding to 'one' (59).

(59) To leave$_{\varphi=\text{'one'}}$ now would be a big mistake.

4.11.4 The pragmatic pathway

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

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

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

4.11.5 Operators

Operators are processed inside an operator-variable module by linking lexical elements containing operator features with a scope-marker that determines the propositional scope for that particular operator. The operation is triggered when narrow semantics sends a lexical element containing an operator feature for the operator-variable module for interpretation. The scope marker is closest lexical head inside the upward path with [OP:F][FIN]. The finite operator cluster [OP:F][FIN] is created during reconstruction. The relevant configuration is illustrated by (60).

(60) Who does$_{wh,\text{fin}}$ John admire (who$_{wh}$)?

4.11.6 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 (61), *him* cannot denote a too local antecedent (62), and R-expressions such as proper names must remain free (63).

(61) a. John$_1$ admires himself$_{1,*2}$.
b. *John's$_1$ sister admires himself$_1$

(62) a. John$_1$ admires him$_{*1,2}$
b. John's$_1$ sister admires him$_{1,2}$

(63) a. He$_1$ admires John$_{*1,2}$.
b. His$_{1,2}$ sister hates John$_1$.

Binding theory is concerned with the conditions that regulate these properties. Since assignments are computed at the language-cognition interface, whatever mechanism drives binding must regulate what takes place there. 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.

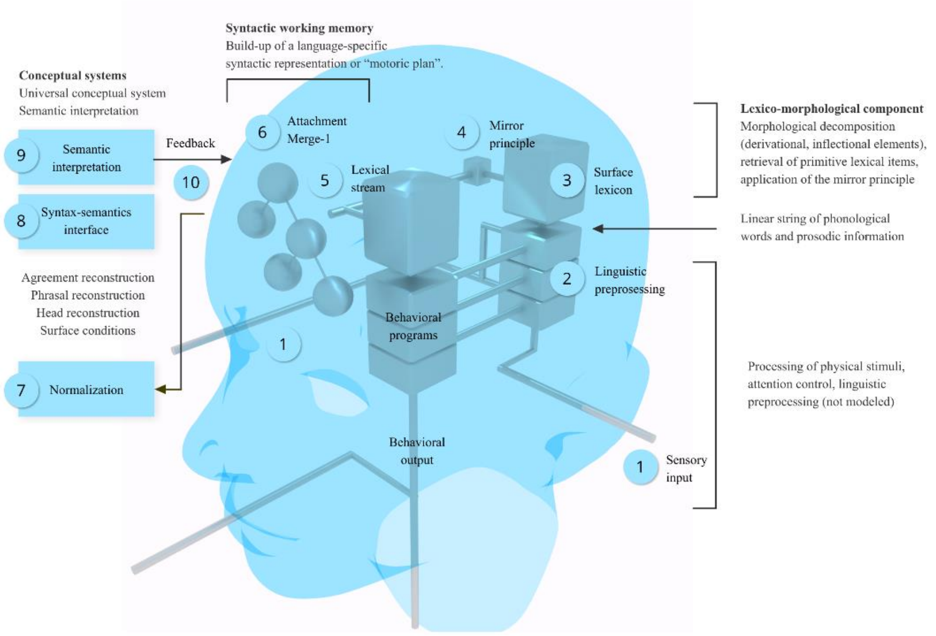
5 Performance

5.1 Human and engineering parsers

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

5.2 Mapping between the algorithm and brain

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



Syntactic working memory
Build-up of a language-specific syntactic representation or "motoric plan".
Conceptual systems
Universal conceptual system
Semantic interpretation
9 Semantic interpretation
Feedback
10
8 Syntax-semantics interface
Agreement reconstruction
Phrasal reconstruction
Head reconstruction
Surface conditions
7 Normalization
6 Attachment Merge-1
5 Lexical stream
4 Mirror principle
3 Surface lexicon
2 Linguistic preprocessing
1
Behavioral programs
Behavioral output
1 Sensory input
Lexico-morphological component
Morphological decomposition (derivational, inflectional elements), retrieval of primitive lexical items, application of the mirror principle
Linear string of phonological words and prosodic information
Processing of physical stimuli, attention control, linguistic preprocessing (not modeled)

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: 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 (64), 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.

(64)

a. The horse raced past the barn.

b. The horse raced past the barn fell.

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

```
1    the + horse
2    [the horse]
     [the horse] + T
3    [[the horse] T]
     [[the horse] T] + race
4    [[the horse] T(V)]
     [[the horse] T(V)] + past
5    [[the horse] [T(V) past]]
     [[the horse] [T(V) past]] + the
6    [[the horse] [T(V) [past the]]]
     [[the horse] [T(V) [past the]]] + barn
7    [[the horse] [T(V) [past [the barn]]]]
     [[the horse] [T(V) [past [the barn]]]] + T
8    [[the horse] [T(V) [past [[the barn] T]]]]
     [[the horse] [T(V) [past [[the barn] T]]]] + fell
9    [[the horse] [T(V) [past [the [barn T]]]]]
     [[the horse] [T(V) [past [the [barn T]]]]] + fell
10   [[the horse] [T(V) [[past [the barn]] T]]]
     [[the horse] [T(V) [[past [the barn]] T]]] + fell
11   [[[the horse]:11 [T [__:11 [race [past [the barn]]]]]] T]
     [[[the horse]:11 [T [__:11 [race [past [the barn]]]]]] T] + fell
12   [[[the horse]:12 [T [__:12 race]]] past]
     [[[the horse]:12 [T [__:12 race]]] past] + the
13   [[[the horse]:12 [T [__:12 race]]] [past the]]
     [[[the horse]:12 [T [__:12 race]]] [past the]] + barn
14   [[[the horse]:12 [T [__:12 race]]] [past [the barn]]]
     [[[the horse]:12 [T [__:12 race]]] [past [the barn]]] + T
15   [[[the horse]:12 [T [__:12 race]]] [past [[the barn] T]]]
     [[[the horse]:12 [T [__:12 race]]] [past [[the barn] T]]] + fell
16   [[[the horse]:12 [T [__:12 race]]] [past [the [barn T]]]]
     [[[the horse]:12 [T [__:12 race]]] [past [the [barn T]]]] + fell
17   [[[the horse]:12 [T [__:12 race]]] [[past [the barn]] T]]
     [[[the horse]:12 [T [__:12 race]]] [[past [the barn]] T]] + fell
18   [[[[the horse]:12 [T [__:12 race]]] <past [the barn]>] T]
     [[[[the horse]:12 [T [__:12 race]]] <past [the barn]>] T] + fell
19   [the [horse T]]
     [the [horse T]] + race
20   [the [horse T(V)]]
     [the [horse T(V)]] + past
21   [the [horse [T(V) past]]]
     [the [horse [T(V) past]]] + the
22   [the [horse [T(V) [past the]]]]
     [the [horse [T(V) [past the]]]] + barn
23   [the [horse [T(V) [past [the barn]]]]]
     [the [horse [T(V) [past [the barn]]]]] + T
24   [the [horse [T(V) [past [[the barn] T]]]]]
     [the [horse [T(V) [past [[the barn] T]]]]] + fell
25   [the [horse [T(V) [past [the [barn T]]]]]]
     [the [horse [T(V) [past [the [barn T]]]]]] + fell
26   [the [horse [T(V) [[past [the barn]] T]]]]
     [the [horse [T(V) [[past [the barn]] T]]]] + fell
27   [[the [horse [T [race [past [the barn]]]]]] T]
     [[the [horse [T [race [past [the barn]]]]]] T] + fell
28   [[the [horse [T [race [past the]]]]] barn]
     [[the [horse [T [race [past the]]]]] barn] + T
29   [[the [horse [T [race [past the]]]]] [barn T]]
     [[the [horse [T [race [past the]]]]] [barn T]] + fell
30   [[the [horse [T [race past]]]] the]
     [[the [horse [T [race past]]]] the] + barn
31   [[the [horse [T [race past]]]] [the barn]]
     [[the [horse [T [race past]]]] [the barn]] + T
32   [[the [horse [T [race past]]]] [[the barn] T]]
     [[the [horse [T [race past]]]] [[the barn] T]] + fell
33   [[the [horse [T [race past]]]] [the [barn T]]]
     [[the [horse [T [race past]]]] [the [barn T]]] + fell
34   [[the [horse [T race]]] past]
     [[the [horse [T race]]] past] + the
35   [[the [horse [T race]]] [past the]]
     [[the [horse [T race]]] [past the]] + barn
36   [[the [horse [T race]]] [past [the barn]]]
     [[the [horse [T race]]] [past [the barn]]] + T
37   [[the [horse [T race]]] [past [[the barn] T]]]
     [[the [horse [T race]]] [past [[the barn] T]]] + fell
38   [[the [horse [T race]]] [past [the [barn T]]]]
     [[the [horse [T race]]] [past [the [barn T]]]] + fell
39   [[the [horse [T race]]] [[past [the barn]] T]]
     [[the [horse [T race]]] [[past [the barn]] T]] + fell
```

```
40   [the horse]
     [the horse] + T/prt
41   [the [horse T/prt]]
     [the [horse T/prt]] + race
42   [the [horse T/prt(V)]]
     [the [horse T/prt(V)]] + past
43   [the [horse [T/prt(V) past]]]
     [the [horse [T/prt(V) past]]] + the
44   [the [horse [T/prt(V) [past the]]]]
     [the [horse [T/prt(V) [past the]]]] + barn
45   [the [horse [T/prt(V) [past [the barn]]]]]
     [the [horse [T/prt(V) [past [the barn]]]]] + T
46   [the [horse [T/prt(V) [past [[the barn] T]]]]]
     [the [horse [T/prt(V) [past [[the barn] T]]]]] + fell
47   [the [horse [T/prt(V) [past [the [barn T]]]]]]
     [the [horse [T/prt(V) [past [the [barn T]]]]]] + fell
48   [the [horse [T/prt(V) [[past [the barn]] T]]]]
     [the [horse [T/prt(V) [[past [the barn]] T]]]] + fell
49   [the [horse [[T/prt [race [past [the barn]]]] T]]]
     [the [horse [[T/prt [race [past [the barn]]]] T]]] + fell
50   [the [horse [[T/prt [race [past [the barn]]]] [T fell]]]] (<= accepted)
```

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

I do not assume that the backtracking operation visible in the example above is entirely realistic from the psycholinguistic point of view. There are two reasons why it exists. One is that by performing systematic backtracking we let the model to explore the complete parsing tree. If we only generated the first solution, spurious secondary solutions might escape our attention. It is not untypical that the model finds ungrammatical and/or wrong secondary solutions, revealing that it is using a wrong competence model. The second reason is that backtracking gives us important information concerning the model's performance. We can compare the amount of computational resources spend in processing different types of constructions and/or different algorithmic solutions. In addition, realistic language comprehension by the human brain in connection with canonical sentences is seldom subject to any garden pathing, so we can at least aim for a model that finds the correct solution immediately, at least in those cases.[26]

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

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

inflectional features, they are all stored in the same memory system and then inserted inside the next lexical item as a set of features (order is ignored).

5.3.2 Connectness

Connectness refers to the property that all incoming linguistic material is attached to one phrase structure that connects everything together. There never occurs a situation where the syntactic working memory holds two or more phrase structures that are not connected to each other by means of some grammatical dependency. Adjunct structures are attached to their host structures loosely: they are geometrical constituents inside their host constructions, observing connectness, but invisible to many computational processes applied to the host (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. The current parsing model is serial in the sense that the predicted cognitive cost is computed in this way by adding the cognitive costs from each individual operation together. We could include parallel processing into the model by calculating the cognitive cost differently, i.e., not adding up the cost of computational operations that are predicted to be performed in parallel. Another complicating factor is that in some cases the implementation order does not seem to matter. We could simply *assume* that the processing is implemented by utilizing parallel processes. Despite these concerns, most computational operations implemented by the linear phase model must be executed in an exact specific order in order to derive empirically correct results. Therefore, for the most part the model assumes that language processing is serial.

5.3.4 Locality preference

Locality preference is a heuristic principle of the human language comprehension which requires that local attachment solutions are preferred over nonlocal ones. A local attachment solution means the lowest right edge in the existing partial phrase structure representation. Thus, the preposition phase *with a telescope* in (65) is first attached to the lowest possible solution, and only if that solution fails, to the nonlocal node.

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

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

5.3.8 Conflict resolution and weighting

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

5.4 Measuring predicted cognitive cost of processing

Processing of words and whole sentences is associated with a predicted cognitive cost, measured in milliseconds. This is done by associating each word with a preprocessing time depending on its phonetic length (currently 25ms per phoneme) and then summing predictive cognitive costs from each computational operation together. Most operations are currently set to consume 5ms, but the user can define these in a way that best agrees with experimental and neurobiological data. Reactivation of a constituent that is not inside the active working memory consumes 500ms. The resulting timing information will be visible in the log files and in the resource outputs. The processing time consumed by each sentence is simply the sum of the processing time of all of its words. A useful metric in assessing the relative processing difficulty of any given sentence is to calculate the mean predicted cognitive processing time per each word (total time / number of words). This metric takes sentence length into account. Resource consumption is summarized in the resource output file (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 *plausbility_metrics.py* which 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 ordered by applying the selected locality preference algorithm, which provides the default ordering, and then by applying all other ranking principles such as lexical anticipation. Finally, the concatenated list of ordered nodes + nodes not active in the working memory and not processed are returned. The parser uses the ordered list to organize the parsing derivation. Notice that the inactive nodes that are not in the active working memory must be part of the list as the parser must be able to explore them if everything else fails, but since the plausibility module does not process them, their order is independent of the incoming word and the partial phrase structure representation currently being constructed.

6 Inputs and outputs

6.1 Installation and use

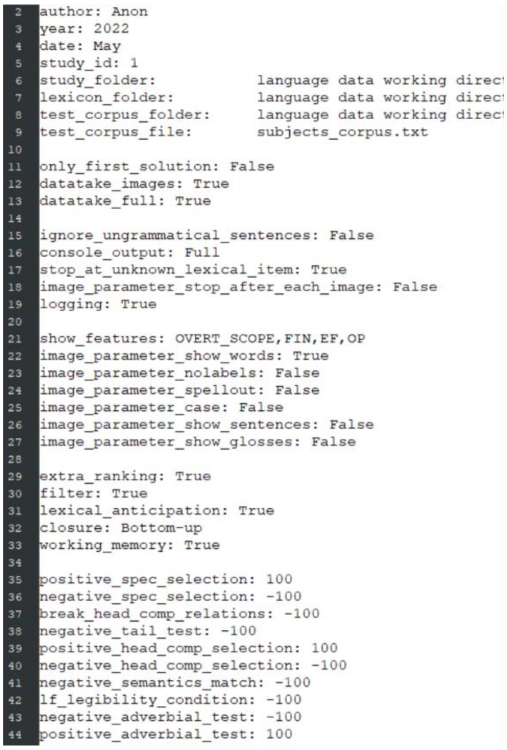
The program is installed by cloning the whole directory from Github.

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

The user must define a folder in the local computer where the program is cloned. This folder will then become the root folder for the project. The root folder will contain at least the following subfolders: */docs* (documentation, such as this document), */language data working directory* (where each individual study is located) and */lpparse* (containing the actual Python modules). In order to run the program the user must have Python (3.x) installed in the local computer and that installation must be specified in the windows path-variable. Refer to Python installation guide for how to accomplish this. The details depends on the operating system. The program can then be used by opening a command prompt into the program root folder and writing

```
python lpparse
```

into the command prompt, which will parse all the sentences from the designated test corpus file in a study folder. Each trial run that is launched by the above command involves a host of internal parameters that the user can configure. These involve things such as where is the lexicon, test corpus, what heuristic principles should be used, and others. The information can be provided in several ways. One way is to provide it inside a configuration file called *config_study.txt*. If the parser is launched without any parameters, as in the example above, it will try to locate this file from the installation directory. Usually this file is present in any version currently being developed. If the file is not found, default values will be used. Any parameter that is defined in the *config_study.txt* can be defined as an input parameter to the program, which will overwrite any specifications found from the configuration file. This makes it possible to control the execution of the script form an external source, say from an external program that one might want to use to organize scripts that perform several studies. Figure 35 shows the contents of the configuration as it exists in my local machine at the time of writing.



```
author: Anon
year: 2022
date: May
study_id: 1
study_folder:           language data working direc
lexicon_folder:         language data working direc
test_corpus_folder:     language data working direc
test_corpus_file:       subjects_corpus.txt

only_first_solution: False
datatake_images: True
datatake_full: True

ignore_ungrammatical_sentences: False
console_output: Full
stop_at_unknown_lexical_item: True
image_parameter_stop_after_each_image: False
logging: True

show_features: OVERT_SCOPE,FIN,EF,OP
image_parameter_show_words: True
image_parameter_nolabels: False
image_parameter_spellout: False
image_parameter_case: False
image_parameter_show_sentences: False
image_parameter_show_glosses: False

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

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

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

Each line has two fields: the key and a value, separated by semicolon. The key determines the name of the parameter. For example, the key *test_corpus_folder* determines the name of the parameter that defines the folder from where the program tries to find the test corpus file. It is followed by the name of the folder. If a key or parameter is missing, the parameter is not used, or a negative value is assumed. If a parameter is missing that is mandatory for normal operation, such as the test corpus file, the program will attempt to use a default value. If also that strategy fails, an error occur. The same key-value pairs can be given as input arguments to the function from the command prompt. They are given by writing *key*=*value*, that is, key followed by "=" followed by the value. For example, if the user wants to run a test corpus from folder */my_test*, then the following command executes the script with that parameter:

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

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

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

```
python lpparse “test_corpus_folder=my test folder/”
```

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

6.2 General organization

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

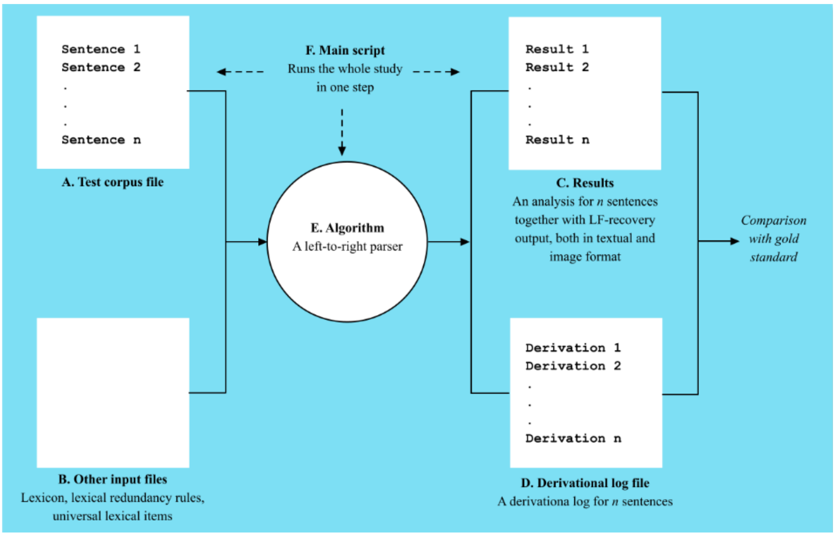

Sentence 1
Sentence 2
.
.
.
Sentence n
A. Test corpus file
F. Main script
Runs the whole study in one step
Result 1
Result 2
.
.
.
Result n
C. Results
An analysis for *n* sentences together with LF-recovery output, both in textual and image format
E. Algorithm
A left-to-right parser
Comparison with gold standard
Derivation 1
Derivation 2
.
.
.
Derivation n
B. Other input files
Lexicon, lexical redundancy rules, universal lexical items
D. Derivational log file
A derivationa log for *n* sentences

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

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

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

The code exists as 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.
scrambling_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.
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.
language_guesser.py	Hosts the code which determines the language used in an input sentence. The constructor will first read the lexicon and extract the languages available there, based on LANG features. The guesser will then determine the language of an input sentence on the basis of its words.
lexical_interface.py	This module reads and processes lexical information. Lexical information is read from the three external files and further processed through a function that applies "parameters". It is stored into a dictionary. Each parser object has its own lexicon that are initialized for each language in the main script.
lexical_stream.py	Defies properties characterizing the lexical stream which "streams" primitive lexical items and inflectional features from the lexico-morphological component into the syntactic component.
LF.py	Processes LF-interface (syntax-semantics interface) objects, with the main role being the checking of LF-legibility.
linear_phase_parser.py	This module defines the parser and its operations (main parse function plus the recursive function called by the former). This is a purely performance module: it reads the input sentence, generates the recursive parse tree, evaluates and stored the results.
local_file_system.py	Handles all I/O behaviors, including console.
SEM_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.
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.
phrase_structure.py	A class that defines the phrase structure objects and the grammatical configurations and relations defined on them.
SEM_control_and_thematic_roles	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.
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 a master lexicon. Once a study is published, however, a copy of the lexical resources used in that study should be stored in connection with the rest of the study-specific materials inside the specific folder.

6.3 Structure of the input files

6.3.1 Test corpus file (any name)

The test corpus file name and location are provided in the *config_study.txt*. Certain conventions must be followed when preparing the file. Each sentence is a linear list of phonological words, separated by white space 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, so 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 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= will be processed).[27]

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 (66).

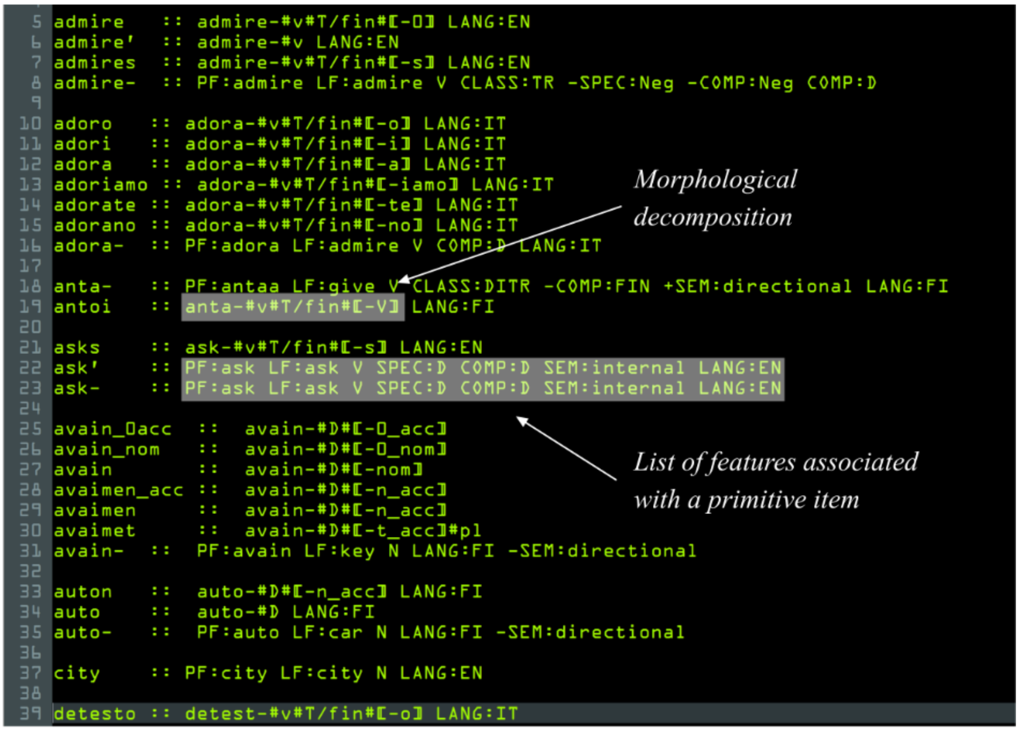
(66) a. John$_1$ met Mary$_2$;
b. He$_{1,3}$ admires her$_{2,4}$.

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

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 (66)a did not end with the semicolon, pronouns *he* and *her* in sentence (b) can no longer refer to them. Conversations can be used to create discourse contexts for test sentences.

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

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



```
admire    :: admire-#v#T/fin#[-0] LANG:EN
admire'   :: admire-#v LANG:EN
admires   :: admire-#v#T/fin#[-s] LANG:EN
admire-   :: PF:admire LF:admire V CLASS:TR -SPEC:Neg -COMP:Neg COMP:D

adoro    :: adora-#v#T/fin#[-o] LANG:IT
adori    :: adora-#v#T/fin#[-i] LANG:IT
adora    :: adora-#v#T/fin#[-a] LANG:IT
adoriamo :: adora-#v#T/fin#[-iamo] LANG:IT
adorate :: adora-#v#T/fin#[-te] LANG:IT
adorano :: adora-#v#T/fin#[-no] LANG:IT
adora-   :: PF:adora LF:admire V COMP:D LANG:IT

anta-    :: PF:antaa LF:give V CLASS:DITR -COMP:FIN +SEM:directional LANG:FI
antoi    :: anta-#v#T/fin#[-V] LANG:FI

asks     :: ask-#v#T/fin#[-s] LANG:EN
ask'     :: PF:ask LF:ask V SPEC:D COMP:D SEM:internal LANG:EN
ask-     :: PF:ask LF:ask V SPEC:D COMP:D SEM:internal LANG:EN

avain_0acc  ::  avain-#D#[-0_acc]
avain_nom   ::  avain-#D#[-0_nom]
avain       ::  avain-#D#[-nom]
avaimen_acc ::  avain-#D#[-n_acc]
avaimen     ::  avain-#D#[-n_acc]
avaimet     ::  avain-#D#[-t_acc]#pl
avain-   ::  PF:avain LF:key N LANG:FI -SEM:directional

auton    ::  auto-#D#[-n_acc] LANG:FI
auto     ::  auto-#D LANG:FI
auto-    ::  PF:auto LF:car N LANG:FI -SEM:directional

city     :: PF:city LF:city N LANG:EN

detesto :: detest-#v#T/fin#[-o] LANG:IT
```

Morphological decomposition

List of features associated with a primitive item

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

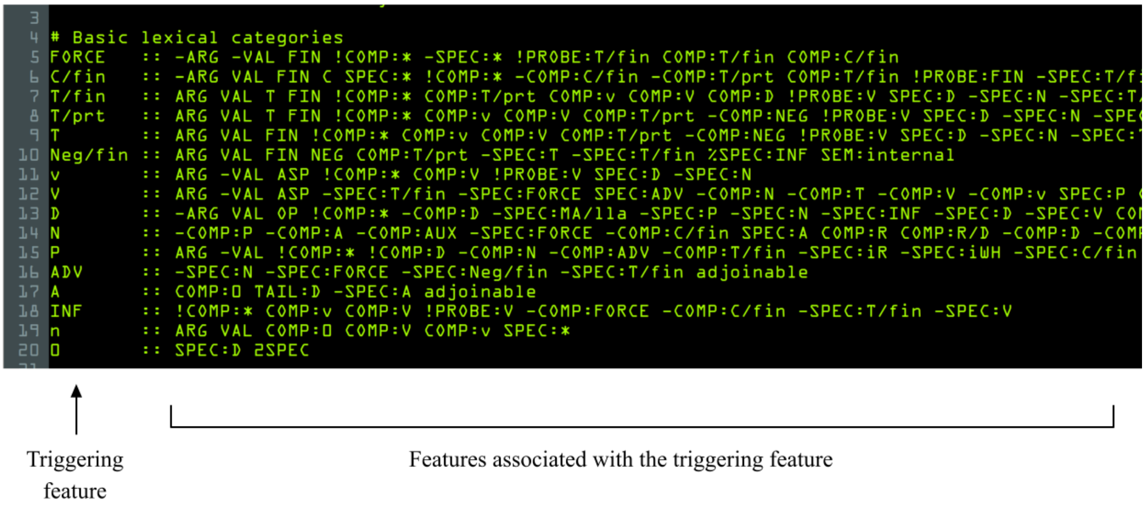
Each line in the lexicon file begins with the surface entry that is matched in the input. This is followed by :: which separates the surface entry from the definition of the lexical item itself. If the surface entry has morphological decomposition, it follows the surface entry and is given in the format '*m*#*m*#*m*#…#*m*' where each item *m* must be found from the lexicon. Symbol # represents morpheme boundary. The individual constituents are thus separated by symbol # which defines the morphological decomposition. If the element designates a primitive (terminal) lexical item, the entry is followed by a list of lexical features. Each lexical feature will be inserted as such inside that lexical item, in the set constituting that item, when it is streamed into syntax. An inflectional feature is designated by the fact that its morphemic decomposition is replaced with symbol “–” or by the word “inflectional.” They are otherwise defined as any other lexical item, namely as a set of features. These features are inserted inside full lexical items during lexical streaming.

A lexical feature is a string, essentially a formal pattern. They do not have further structure and are processed by first-order Markovian operations. The way any given feature reacts inside syntax and semantics is defined by the computations that process these lexical items and the patterns in them.[28] As can be seen from the above screen capture, most features have a ‘type:value’ structure, where the type dictates the system that processes it, value is the input that will generate a specific interpretation.

The file *ug_morphemes.txt* is structured in the same way but contains universal morphemes such as T and v.

6.3.3 Lexical redundancy rules

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



```
# Basic lexical categories
FORCE   :: -ARG -VAL FIN !COMP:* -SPEC:* !PROBE:T/fin COMP:T/fin COMP:C/fin
C/fin   :: -ARG VAL FIN C SPEC:* !COMP:* -COMP:C/fin -COMP:T/prt COMP:T/fin !PROBE:FIN -SPEC:T/f
T/fin   :: ARG VAL T FIN !COMP:* COMP:T/prt COMP:v COMP:V COMP:D !PROBE:V SPEC:D -SPEC:N -SPEC:T
T/prt   :: ARG VAL T FIN !COMP:* COMP:v COMP:V COMP:T/prt -COMP:NEG !PROBE:V SPEC:D -SPEC:N -SPE
T       :: ARG VAL FIN !COMP:* COMP:v COMP:V COMP:T/prt -COMP:NEG !PROBE:V SPEC:D -SPEC:N -SPEC:
Neg/fin :: ARG VAL FIN NEG COMP:T/prt -SPEC:T -SPEC:T/fin %SPEC:INF SEM:internal
v       :: ARG -VAL ASP !COMP:* COMP:V !PROBE:V SPEC:D -SPEC:N
V       :: ARG -VAL ASP -SPEC:T/fin -SPEC:FORCE SPEC:ADV -COMP:N -COMP:T -COMP:V -COMP:v SPEC:P
D       :: -ARG VAL OP !COMP:* -COMP:D -SPEC:MA/lla -SPEC:P -SPEC:N -SPEC:INF -SPEC:D -SPEC:V CO
N       :: -COMP:P -COMP:A -COMP:AUX -SPEC:FORCE -COMP:C/fin SPEC:A COMP:R COMP:R/D -COMP:D -COM
P       :: ARG -VAL !COMP:* !COMP:D -COMP:N -COMP:ADV -COMP:T/fin -SPEC:iR -SPEC:iWH -SPEC:C/fin
ADV     :: -SPEC:N -SPEC:FORCE -SPEC:Neg/fin -SPEC:T/fin adjoinable
A       :: COMP:0 TAIL:D -SPEC:A adjoinable
INF     :: !COMP:* COMP:v COMP:V !PROBE:V -COMP:FORCE -COMP:C/fin -SPEC:T/fin -SPEC:V
n       :: ARG VAL COMP:0 COMP:V COMP:v SPEC:*
0       :: SPEC:D 2SPEC
```

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

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

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

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

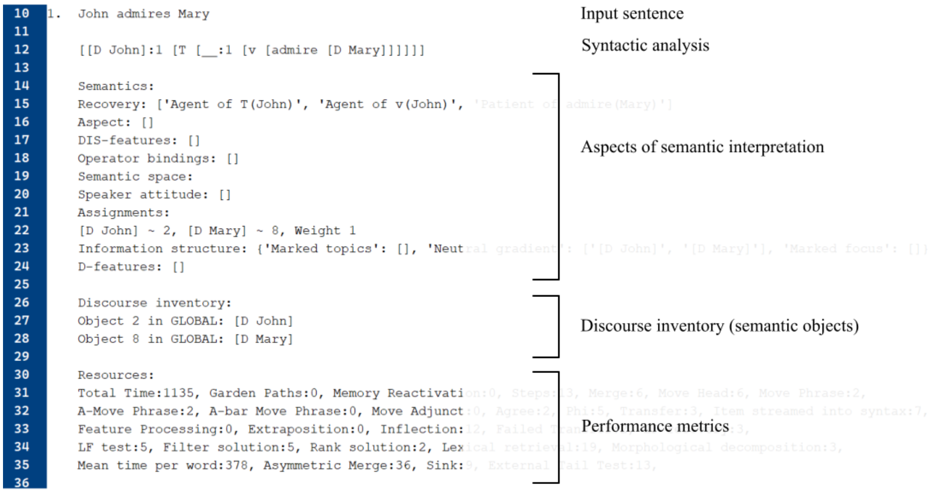
6.3.4 Study parameters (config_study.txt)

It is possible to associate each study with specific study parameters which tell how the parser operates. These parameters are contained in the file *config_study.txt*. The parameters are also stamped on the output files.

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 in *config_study.txt*. 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. 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
11
12     [[D John]:1 [T [__:1 [v [admire [D Mary]]]]]]
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: []
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]
28     Object 8 in GLOBAL: [D Mary]
29
30     Resources:
31     Total Time:1135, Garden Paths:0, Memory Reactivation:0, Steps:13, 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,
33     Feature Processing:0, Extraposition:0, Inflection:12, Failed Tran[illegible]: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
```

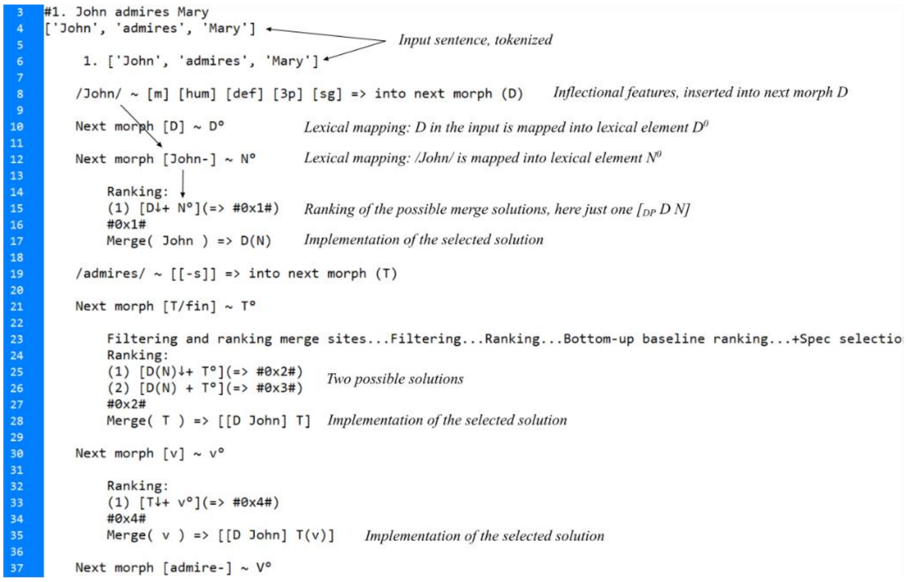
Input sentence
Syntactic analysis
Aspects of semantic interpretation
Discourse inventory (semantic objects)
Performance metrics

Figure 19. Screen capture from a results file.

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

6.4.2 The log file

The derivational log file, created by default by adding *_log* into the name of the test corpus file, contains a more detailed report of the computational steps consumed in processing each sentence in the test corpus and of the semantic interpretation. The log file uses the same numerical identifiers as the results file. In order to locate the derivation for sentence number 1, for example, you would search for string "# 1" from the log file. What type of information is reported in the log file can be decided freely. By default, however, the log file contains information about the processing and morphological decomposition of the phonological words in the input, application of the ranking principles leading into Merge-1, transfer operation applied to the final structure when no more input words are analyzed and many aspects of semantic interpretation. Intermediate left branch transfer operations are not reported in detail. The beginning of a log file is illustrated in Figure 20, with some explanations added.



```
#1. John admires Mary
['John', 'admires', 'Mary']

    1. ['John', 'admires', 'Mary']

    /John/ ~ [m] [hum] [def] [3p] [sg] => into next morph (D)

    Next morph [D] ~ D°

    Next morph [John-] ~ N°

        Ranking:
        (1) [D↓+ N°](=> #0x1#)
        #0x1#
        Merge( John ) => D(N)

    /admires/ ~ [[-s]] => into next morph (T)

    Next morph [T/fin] ~ T°

        Filtering and ranking merge sites...Filtering...Ranking...Bottom-up baseline ranking...+Spec selectio
        Ranking:
        (1) [D(N)↓+ T°](=> #0x2#)
        (2) [D(N) + T°](=> #0x3#)
        #0x2#
        Merge( T ) => [[D John] T]

    Next morph [v] ~ v°

        Ranking:
        (1) [T↓+ v°](=> #0x4#)
        #0x4#
        Merge( v ) => [[D John] T(v)]

    Next morph [admire-] ~ V°
```

Input sentence, tokenized

Inflectional features, inserted into next morph D

Lexical mapping: D in the input is mapped into lexical element D^0

Lexical mapping: /John/ is mapped into lexical element N^0

Ranking of the possible merge solutions, here just one [$_{DP}$ D N]

Implementation of the selected solution

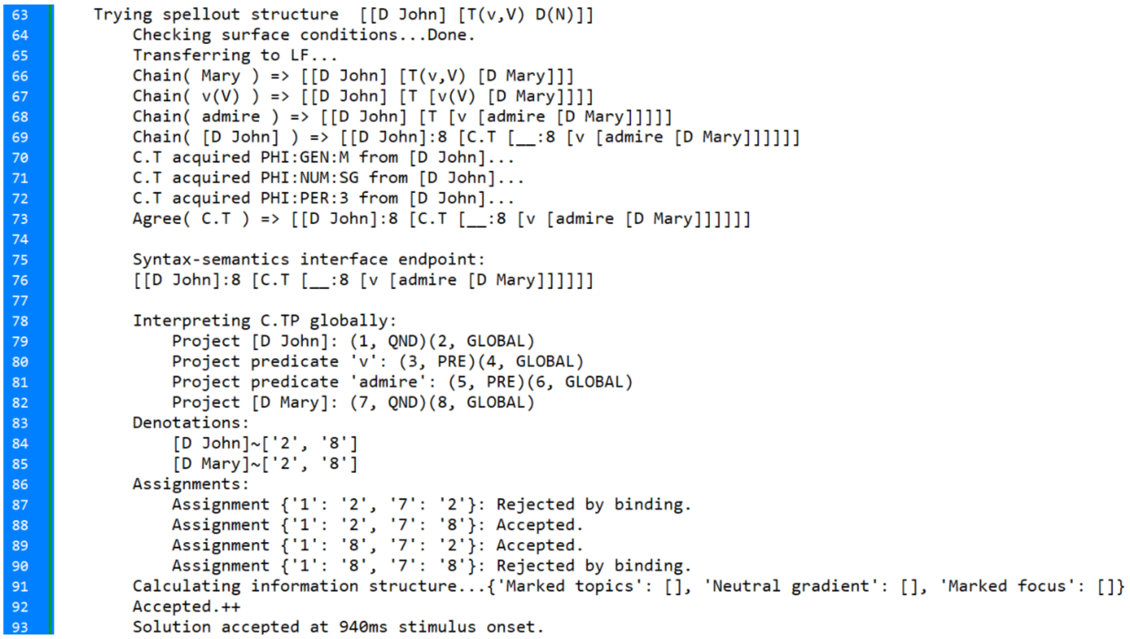
Two possible solutions

Implementation of the selected solution

Implementation of the selected solution

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*.



```
63    Trying spellout structure  [[D John] [T(v,V) D(N)]]
64        Checking surface conditions...Done.
65        Transferring to LF...
66        Chain( Mary ) => [[D John] [T(v,V) [D Mary]]]
67        Chain( v(V) ) => [[D John] [T [v(V) [D Mary]]]]
68        Chain( admire ) => [[D John] [T [v [admire [D Mary]]]]]
69        Chain( [D John] ) => [[D John]:8 [C.T [__:8 [v [admire [D Mary]]]]]]
70        C.T acquired PHI:GEN:M from [D John]...
71        C.T acquired PHI:NUM:SG from [D John]...
72        C.T acquired PHI:PER:3 from [D John]...
73        Agree( C.T ) => [[D John]:8 [C.T [__:8 [v [admire [D Mary]]]]]]
74
75        Syntax-semantics interface endpoint:
76        [[D John]:8 [C.T [__:8 [v [admire [D Mary]]]]]]
77
78        Interpreting C.TP globally:
79            Project [D John]: (1, QND)(2, GLOBAL)
80            Project predicate 'v': (3, PRE)(4, GLOBAL)
81            Project predicate 'admire': (5, PRE)(6, GLOBAL)
82            Project [D Mary]: (7, QND)(8, GLOBAL)
83        Denotations:
84            [D John]~['2', '8']
85            [D Mary]~['2', '8']
86        Assignments:
87            Assignment {'1': '2', '7': '2'}: Rejected by binding.
88            Assignment {'1': '2', '7': '8'}: Accepted.
89            Assignment {'1': '8', '7': '2'}: Accepted.
90            Assignment {'1': '8', '7': '8'}: Rejected by binding.
91        Calculating information structure...{'Marked topics': [], 'Neutral gradient': [], 'Marked focus': []}
92        Accepted.++
93        Solution accepted at 940ms stimulus onset.
```

Figure 21. Part of the derivational log file.

Lines 63-73 contain transfer. They correspond to well-known linguistic processes (head reconstruction, extraposition, adjunct reconstruction, phrasal Ā/A reconstruction, agreement reconstruction and last resort extraposition). The numbers shown in connection with denotations (lines 84-85) 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 79-82).

6.4.3 *Simple logging*

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

6.4.4 *Saved vocabulary*

Each time a study is run, the program takes a snapshot of the surface vocabulary (lexicon) as it stands after all processing has been done (after each sentence has been processed) and saves it into a separate text file with the suffix *_saved_vocabulary.txt*. The reason is because the ultimate lexicon used in each study is synthesized from three sources (language-specific lexicon, universal morphemes and lexical redundancy rules) and thus involves computations and assumptions whose output the user might want to verify. Notice that the complete feature content of each terminal element that occurs in any output solution is stored into the log file together with the solution (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.

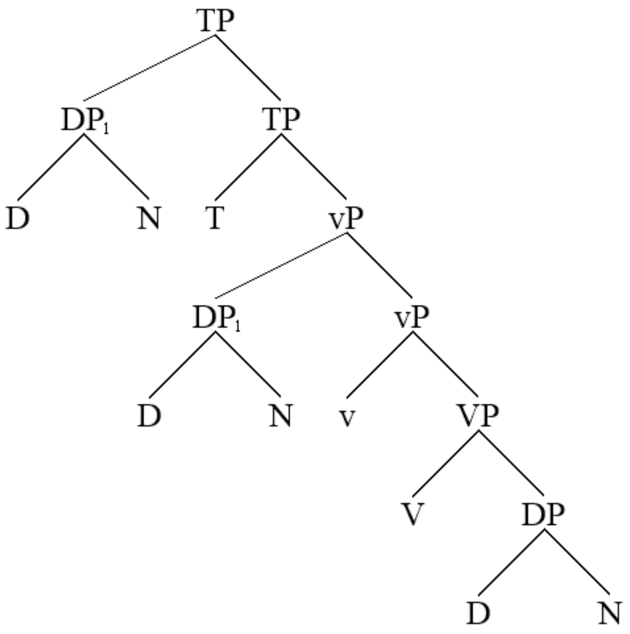


Figure 23. A simple phrase structure image generated by the algorithm for a simple transitive clause *John admires Mary*.

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). The content of these images is controlled by several parameters provided in the *config_study.txt* file and are as follows. Parameters *show_features* provides a list of lexical features that the user wants to show on the three. For example, if this list contains the feature FIN, then all finite heads will show [FIN] below them. *Show_words* will show the phonological entries provided for each lexical element. *Spellout* will draw images also for the spellout structures before any transfer operations had takes place. *Case* will add case features to the lexical items. *Show_sentences* will stamp the input sentence into the image. *Show_glosses* will add English translation to each lexical item.

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. 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 also contains 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.

6.4.8 Control and thematic roles

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

6.5 Main script

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

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

The first line initializes the 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 the "brain model" for each language. 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 change the brain model on the basis of the language in the input sentence. The computational core of the model has no language-specific parameters; rather, they came into effect when the functional lexicon is processed through lexical redundancy rules. The issue is empirically nontrivial. Currently, it is assumed that only the functional lexicon changes. When the lexicon is loaded, all lexical items that do not have specification for language – usually these are only functional items – will be provided one by using the contextual language parameter. This will trigger language-specific redundancy rules when these rules are applied.

```
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. 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 phrase structure class defined in *phrase_structure.py* defines the phrase structure objects called constituents that are manipulated at each stage of the processing pipeline.

7.1.2 Lexical items

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

7.1.3 Phrase structure geometry

Each constituent α can have the *left daughter constituent* and the *right daughter constituent*. Their *mother* will be α.

```
self.left = left_constituent
self.right = right_constituent
if self.left:
    self.left.mother = self
if self.right:
    self.right.mother = self
```

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

```
def complex(self):
    return self.right and self.left
```

A constituent is complex if it is not primitive.

```
def primitive(self):
    return not self.complex()
```

It follows that a constituent that has zero *or one immediate daughter* is primitive, creating a three-way classification: complex versus primitive constituents, where the latter is further divided into terminal (with zero daughters) and nonterminal constituents (with one daughter). Nonterminal nonphrasal constituents represent *complex heads* (Section 4.8.5). Only phrasal constituents are in the domain of phrasal syntactic rules. A complex constituent can have a *left constituent* and a *right constituent*. These notions are defined as follows.

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


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

We can then define several other relations:

```
def bottom(x):
    while not x.primitive():
        x = x.right
    return x

def top(x):
    while x.mother:
        x = x.mother
    return x

def grandmother(self):
    if self.mother.mother:
        return self.mother.mother

def aunt(self):
    if self.mother:
        return self.mother.sister()
```

7.1.4 Complex heads and affixes

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

```
def has_affix(self):
    return self.right and not self.left


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
```

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.

7.1.5 Sisters

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
    return self.mother.left
```

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

```
def sister(x):
    while x.mother:
        if x.is_left():
            if not x.geometrical_sister().adjunct:
                return x.geometrical_sister()
            else:
                x = x.mother
        if x.is_right():
            if not x.adjunct:
                return x.geometrical_sister()
            else:
                return None
```

This definition ignores invisible (adjoined) right constituents.

7.1.6 Proper selected complement

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

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

def proper_selected_complement(self):
    if self.primitive():
        return self.right_sister()
```

7.1.7 Heads (labels), maximal projection, container

The recursive labeling algorithm is provided informally in (67).

(67) Labeling

Suppose α is a complex phrase. Then

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

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

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

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

In Python:

```
def head(self):
    if self.is_primitive():
        return self
    if self.left.is_primitive():
        return self.left
    if self.right.adjunct:
        return self.left.head()
    if self.right.is_primitive():
        return self.right
    return self.right.head()
```

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

```
def max(self):
    x = self
    while x.mother and x.mother.head() == self:
        x = x.mother
        return x
```

container and *inside*

```
def container(self):
    if self.mother:
        return self.mother.head()

def inside(self, head):
    return self.head() == head
```

where maximum projection refers to the highest node dominating α that has the same head as α, and container refers to the head of α's mother. Finally, α is *inside* β if and only if β is the head of α.

7.1.8 Minimal search, geometrical minimal search and upstream search

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

```
def __next__(self):
    if not self.nn:
        raise StopIteration
    current = self.nn
    if self.nn.is_primitive():
        self.nn = None
        return current
    elif self.nn.head() == self.nn.right.head() or self.nn.left.select_right():
        self.nn = self.nn.right
    else:
        self.nn = self.nn.left
    return current.left

def __iter__(self):
    self.nn = self
    return self
```

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

```
def bare_minimal_search(self):
     return [const for const in self].
```

This function does not exist as a separate item, because it can be created simply by creating the iterator ([*const for const in self*]) when needed. The minimal search function in the current implementation is slightly more abstract and defined as follows:

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

It has two additional properties: selection function and sustain condition. The former selects items from the bare minimal search based on the condition given in the input (*if selection_condition(const)*), while the latter can be used to define an intervention or termination that halts the search when the condition becomes true (*sustain_condition*). Both are provided as lambda functions. If no functions are provided, all constituents are returned and the search terminates only when a primitive constituent is reaches as defined in the __*next*__ function.

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

```
    def geometrical_minimal_search(x):
        search_list = [x]
        while x.complex() and x.right:
            search_list.append(x.right)
            x = x.right
        return search_list
```

7.1.9 Upward paths

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

```
def upward_path(self):
    upward_path = []
    x = self.mother
    while x:
        if not x.right.adjunct and x.left.head() != self:
            upward_path.append(x.left)
        x = x.mother
    return upward_path
```

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

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

We take all elements β from the upward path as long as the label of the mother of β is α.

7.1.10 Cyclic Merge

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

```
def __init__(self, left_constituent=None, right_constituent=None):
    self.left = left_constituent
    self.right = right_constituent
    if self.left:
        self.left.mother = self
    if self.right:
        self.right.mother = self
    self.mother = None
    self.features = set()
    self.active_in_syntactic_working_memory = True
    self.morphology = ''
    self.internal = False
    self.adjunct = False
    self.incorporated = False
    self.find_me_elsewhere = False
    self.identity = ''
    self.rebaptized = False
    self.stop = False
    self.nn = None
    self.x = 0
    self.y = 0
    if left_constituent and left_constituent.adjunct 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.[29]

7.1.11 Countercyclic Merge-1

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

(68)

a. [X...[$_{\gamma}$α β]...Y]

b. [X...[$_{\gamma}$β α]...Y]

Constituent γ then replaces α in the phrase structure, with the phrase structural relations updated accordingly. Both Merge to the right and left, and both countercyclically and by extending the structure, are allowed. The range of options is compensated by the restricted conditions under which each operation 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

[29] 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 [$_{\alpha}$ β] takes advantage of this option: here [$_{\alpha}$ β] 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" [α β]$_{F}$ which would be complex constituents associated with (lexical) feature bundles F.

snapshot of the local structure around α (its mother and position in the left-right axis), the second creates [α β] or [β α] (*asymmetric_merge*), and the third (*substitute*) substitutes α with the new constituent [α β] by using local constituent relations recorded by the first operation.

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

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

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

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

7.1.12 Remove

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

```
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
```

7.1.13 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
```

7.1.14 Feature checking

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

```
def check(self, feature_set):
    return feature_set & self.head().features == feature_set

def check_some(self, feature_set):
    return feature_set & self.head().features
```

7.1.15 *Probe-goal: probe(label, goal_feature)*

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

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

The operation searches for feature G by using minimal search. The probe-goal mechanism implements nonlocal selection. One example of nonlocal selection is the relationship between D and N. Every D must be paired with an N, but the relationship can be intervened by a number of other functional heads such as Q, Num and others.

7.1.16 *Tail-head relations*

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

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

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

```
def tail_condition(self, tset):
    if self.max() and self.max().container() and (self.max().container().check(tset) or
       (self.max().mother.sister() and self.max().mother.sister().check(tset))):
        return True
    if self.referential() or self.preposition():
        for m in (affix for node in self.upward_path() if node.primitive() for affix in
           node.get_affix_list()):
            if m.check_some(tset):
                return m.check(tset)
```

The weak tail condition defines an *intervention effect*: if a node checks some features in the target set (*if m.check_some(tset)*), then the condition is true if they are all checked, false otherwise. The implication is that also negative features must be checked in clusters.

7.1.17 Abstractions

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

```
def adverbial(self):
    return self.check({'Adv'})

def force(self):
    return self.check({'FORCE'})

def finite(self):
    return self.check_some({'Fin', 'T/fin', 'C/fin'})

def copula(self):
    return self.check({'COPULA'})

def finite_C(self):
    return self.check({'C/fin'})

def relative(self):
    return self.check({'REF'})

def nonfinite(self):
    return self.check({'Inf'})

def concept_operator(self):
    return self.concept() and {feature for feature in self.features if feature[:2] == 'OP'}

def finite_left_periphery(self):
    return self.finite() and self.check_some({'T', 'C'})

def finite_tense(self):
    return self.check({'T/fin'}) or (self.finite() and self.check({'T'}))

def contains_finiteness(self):
    return self.contains_features({'Fin'})

def referential(self):
    return self.check_some({'φ', 'D'})

def preposition(self):
    return self.check({'P'})

def floatable(self):
    return not self.check({'-float'})

def SEM_internal_predicate(self):
    return self.check({'SEM:internal'})

def SEM_external_predicate(self):
    return self.check({'SEM:external'})

def non_scopal(self):
    return self.check_some({'Inf', 'P', 'D', 'φ'})

def expresses_concept(self):
    return self.check_some({'N', 'Neg', 'P', 'D', 'φ', 'A', 'V', 'Adv', 'Q', 'Num', '0'}) and not
self.check({'T/prt', 'COPULA'})

def unrecognized_label(self):
    return self.check_some({'CAT:?', '?'})

def predicate(self):
    return self.primitive() and self.check({'ARG'}) and not self.check({'-ARG'})

def adverbial_adjunct(self):
    return self.adverbial() or self.preposition()

def is_adjoinable(self):
    return self.adjunct or (self.head().check({'adjoinable'}) and not self.head().check({'-
adjoinable'}))

def clitic(self):
    return self.check({'CL'}) or self.head().check({'CL'}) and not self.head().has_affix() and
self.head().internal

def concept(self):
    next((x for x in self.get_affix_list() if x.expresses_concept()), False)

def semantic_complement(self):
    return self.proper_selected_complement() and not
self.semantic_match(self.proper_selected_complement())

def selected_by_SEM_internal_predicate(self):
    return self.selector() and self.selector().SEM_internal_predicate()

def selected_by_SEM_external_predicate(self):
    return self.selector() and self.selector().SEM_external_predicate()

def isolated_preposition(self):
    return self.preposition() and self.sister() and self.sister().is_primitive()

def adjoinable(self):
    return self.complex() and not self.find_me_elsewhere and self.head().get_tail_sets() and
self.check({'adjoinable'}) and not self.check({'-adjoinable'})

def legitimate_criterial_feature(self):
    return self.referential() and not self.relative() and self.mother and
self.mother.contains_features({'REL'}) and not self.mother.contains_features({'T/fin'})

def interpretable_adjunct(self):
    return self.referential() and self.max() and self.max().adjunct and self.max().is_right() and
self.max().mother and self.max().mother.referential()

def word_internal(self):
    return self.bottom().bottom_affix().internal

def impossible_sequence(self, w):
    return self.primitive() and 'T/fin' in self.head().features and 'T/fin' in w.features

def is_word_internal(self):
    return self.mother and self.sister() and self.sister().primitive() and self.sister().internal
```

7.2 Transfer (transfer.py)

7.2.1 A comment on the current version (14.2)

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

7.2.2 Generalized transfer

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

```
def execute_sequence(self, ps):
    self.reconstruct(ps, self.instructions['Head'])
    self.reconstruct(ps, self.instructions['Feature'])
    self.reconstruct(ps, self.instructions['Extraposition'])
    ps = self.scrambling_module.reconstruct(ps)
    self.reconstruct(ps, self.instructions['Phrasal'])
    self.reconstruct(ps, self.instructions['Agree'])
    self.reconstruct(ps, self.instructions['Last Resort Extraposition'])
    return ps
```

The general form of each step is defined by *reconstruct* which takes two arguments, the starting node of each cycle, usually the bottom right node, and *instructions*. The scrambling module has different form; it has not been unified with the rest of the transfer operations. See 7.2.8. The *instructions* parameter is a dictionary which contains lambda-functions providing the conditions and operations that are observed and executed during each reconstruction cycle, depending on the type of reconstruction.[30] The reconstruction cycle is defined as follows.

```
def reconstruct(self, probe, inst):
    x = probe.bottom()
    while x:
        if inst['need repair'](x):
            inst['repair function'](x, self, inst)
        x = x.move_upwards()
```

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

7.2.3 Chain creation (all chains)

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

[30] Separating the conditions from the general reconstruction function allows one to eliminate enormous amount of repetition that resulted from the assumption that each cycle was based on a separate function, as was the case in earlier versions. The drawback is that whenever the general reconstruction function is modified, the new version must be tested against a dataset that has at least the standard examples of each reconstruction cycle.

```
def create_chain(self, transfer, inst):
    for i, target in enumerate(self.select_objects_from_edge(inst)):
        inst, target = target.prepare_chain(self, inst, i > 0, target.scan_operators(), transfer)
        self.form_chain(target, inst)
        transfer.brain_model.consume_resources(inst['type'], self)
        if inst['need repair'](target):
            target.create_chain(transfer, inst)
```

The function is called by the probe head H (*self*). The iteration at the beginning selects relevant objects from the edge of H: the head itself in the case of head chain, phrasal specifiers in the case of phrasal chains. There can be several such elements, as in the case of Italian left periphery, for example. The selected items are called *targets*. Next, the chain is prepared (see below) and then created by the *form_chain* function for the *target* element designated by the iteration. If the reconstructed item needs further chain repair, the same function is called recursively. A complex head that hosts several affixes gets cleaned up completely, since only the last step creates a primitive head. This recursion will later handle successive-cyclic phrasal movement.

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

```
def prepare_chain(self, probe, inst, new_head_needed, op_features, transfer):
    if inst['type'] == 'Phrasal Chain':
        if not op_features and inst['last resort A-chain conditions'](self):
                inst['selection'] = lambda x: x.has_vacant_phrasal_position()
                inst['legible'] = lambda x, y: True
        elif new_head_needed and (op_features or self.unlicensed_specifier()):
                probe = self.sister().merge_1(transfer.access_lexicon.PhraseStructure(), 'left').left
            probe.features |=
transfer.access_lexicon.apply_parameters(transfer.access_lexicon.apply_redundancy_rules({'OP:_'} |
self.checking_domain('OP*' in op_features).scan_operators() | probe.add_scope_information()))
        return inst.copy(), self.copy_for_chain(transfer.brain_model.babtize())
```

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

```
def form_chain(self, target, inst):
    for head in self.search_domain().minimal_search(inst['selection'], inst['sustain']):
        if head.test_merge(target, inst['legible'], 'left'):
            break
        target.remove()
    else:
        if not head.test_merge(target, inst['legible'], 'right'):
            target.remove()
            self.sister().merge_1(target, 'left')
```

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

7.2.4 Head chain

A head chain is triggered when the reconstruction cycle finds a complex head X(Y). Head Y is targeted inside X and moved downstream by minimal search (7.1.8) until a position is found where it can be selected.[31] The chain creation algorithm is called recursively if Y itself is a complex head.

7.2.5 A-chains

A-chains constitute a last resort option when an operator Ā-chain cannot be created and if the target satisfies conditions for the A-chain. The decision is currently based on whether the reconstructed specifier is an operator or not. The target phrase is merged to the first vacant position found by minimal search (7.1.8). A vacant position means either the complement of the bottom right node ([XP$_1$...[Y __$_1$]]) or a position between two heads that does not contain other phrases ([XP$_1$...[Y __$_1$ [Z...]]]).[32] If the target element does not satisfy the conditions for A-chains, it is not reconstructed. It may be reconstructed by a later operation. Head reconstruction and scrambling reconstruction are applied before A-chains.

7.2.6 Ā-chains

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

7.2.7 Agree

A head triggers Agree during reconstruction if it has the edge feature [EF]. The operation tries to value the unvalued φ-features of the probe H by locating suitable goals first from within the sister (*Agree_from_sister*) and then, if unvalued features remain, from the edge of H (*Agree_from_edge*).

```
def Agree(self, transfer):
     [...]
        goal, phi = self.Agree_from_sister()
            [...]
```

[31] This version does not generate long head movement, which will be added later. Several previous versions of the algorithms were LHM-compliant, but they used a separate head chain algorithm.

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

```
            for p in phi:
                self.value(goal, p)
            if not self.is_unvalued():
                return

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

The valuation function (*value*) will recognize feature clashes and marks them as bad (later we will rely on feature mismatch at LF), which accounts for agreement errors.

7.2.8 Scrambling (scrambling_reconstruction.py)

Scrambling reconstructions has not been unified with the generalized reconstruction function. It performs symmetric 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 constituent in ps.symmetric_minimal_search(lambda x: x.trigger_adjunct_reconstruction(),
     lambda x: x.is_right()):
        self.reconstruct_scrambled_item(constituent)
    return ps.top()
```

The function locates nodes requiring scrambling reconstruction and reconstructs them. Symmetric minimal search returns both A and B under [A B]. Reconstruction is implemented by the following function.

```
def reconstruct_scrambled_item(self, target):
    [...]
    starting_point = target.container()
    virtual_test_item = target.copy()
    local_tense_edge = target.local_tense_edge()
    for node in local_tense_edge.minimal_search(lambda x: x == x, lambda x: self.sustain_condition(x,
target, local_tense_edge)):
        self.merge_floater(node, virtual_test_item)
        if virtual_test_item.valid_reconstructed_adjunct(starting_point):
            virtual_test_item.remove()
            dropped_floater = self.copy_and_insert(node, target)
    [...]
```

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

7.2.9 Adjunct promotion (adjunct_constructor.py)

Adjunct promotion is an operation that changes the status of a phrase structure from non-adjunct into an adjunct, thus it transfers the constituent from the "primary working space" into the "secondary working space." Decisions concerning what will be an adjunct and non-adjunct cannot be made in tandem with consuming words from the input. The operation is part of transfer. If the phrase targeted by the operation is complex, the operation is trivial: the whole phrase structure is moved. If the targeted item is a primitive head, then there is an extra concern: how much of the surrounding structure should come with the head?

```
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)
    else:
        self.externalize(head.mother)

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 LF-legibility (LF.py)

7.3.1 Overall

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

```
        self.LF_legibility_tests = [self.selection_test,
                                    self.projection_principle,
                                    self.head_integrity_test,
                                    self.feature_conflict_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 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. The test function is boldfaced.

```
    def pass_LF_legibility(self, ps):
        if ps.is_primitive():
            self.complete_edge, self.edge_for_EF = self.create_edges(ps)
            for LF_test in self.active_test_battery:
                result = LF_test(ps)
                if result:
                    log(result)
                    self.error_report_for_user = result
                    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 the specifier and complement selection tests are valid. Selection tests are collected into their own data structure

```
        self.selection_violation_tests = [('-EF:φ', self.selection__negative_SUBJECT_edge),
                                          ('!EF:φ', self.selection__positive_SUBJECT_edge),
                                          ('-EF:*', self.selection__unselective_negative_edge),
                                          ('!1EDGE', self.selection__negative_one_edge),
                                          ('!SEF', self.selection__positive_shared_edge),
                                          ('-SPEC:', self.selection__negative_specifier),
                                          ('!SPEC', self.selection__positive_selective_specifier),
                                          ('-COMP', self.selection__negative_complement),
                                          ('!COMP', self.selection__positive_obligatory_complement)]
```

and are then applies to every lexical feature:

```
def selection_test(self, probe):
    return next((f'\'{probe}\' failed {lexical_feature}' for lexical_feature in
          sorted(for_lf_interface(probe.features))
                for (gate, test) in self.selection_violation_tests
                if lexical_feature.startswith(gate) and
                not test(probe, lexical_feature)), None)
```

The function activates a selection test if the lexical item has the gate feature, as listed in the selection test list above. If the test fails (*not test(probe, lexical_feature)*), the function returns an error report. The tests are defined in the functions in the above data structure.

7.3.2 Edge selection

Selection tests match lexical selection features with the contents of local grammatical context. Each feature type is checked by its own function, although it is clear that a more general system (one function) must be at stake. The complications come from the presence of virtual pronouns (pro-elements) and from the notion of extended subject.

```
   # Feature !EF:φ
    def selection__positive_SUBJECT_edge(self, selected_feature):
        return self.next(self.pro_edge, lambda x: x.referential() and not x.check({'pro_'}) and (not
x.get_tail_sets() or x.extended_subject()))

    # Feature -EF:φ
    def selection__negative_SUBJECT_edge(self, selected_feature):
        return not self.next(self.pro_edge, lambda x: x.referential() and not x.check({'pro_'}) and
(not x.get_tail_sets() or x.extended_subject()))

    # Feature !EF:*
    def selection__unselective_edge(self, selected_feature):
        return self.edge()

    # Feature -EF:*
    def selection__unselective_negative_edge(self, selected_feature):
        return not self.edge()

    # Feature !SEF
    def selection__positive_shared_edge(self, selected_feature):
        return not (not self.licensed_phrasal_specifier()and self.referential_complement_criterion())

    # Feature -SPEC:L
    def selection__negative_specifier(self,  selected_feature):
        return not self.next(self.edge, lambda x: x.check({selected_feature}) and not x.adjunct)

    # Feature !1EDGE
    def selection__negative_one_edge(self, selected_feature):
        return len(self.edge()) < 2

    # Feature !COMP:L
    def selection__positive_obligatory_complement(self, selected_feature):
        return self.selected_sister() and self.selected_sister().check({selected_feature})

    # Feature -COMP:L
    def selection__negative_complement(self, selected_feature):
        return not (self.proper_selected_complement() and
self.proper_selected_complement().check({selected_feature}))

    # Feature [!SEF]
    def referential_complement_criterion(probe):
        return probe.proper_selected_complement() and
(probe.proper_selected_complement().head().referential() or
(probe.proper_selected_complement().head().licensed_phrasal_specifier() and
probe.proper_selected_complement().head().licensed_phrasal_specifier().head().referential()))

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

    def double_spec_filter(self):
        return not self.check({'2SPEC'}) and len({spec for spec in self.edge() if not spec.adjunct})
> 1

    def licensed_phrasal_specifier(self):
        if self.next(self.edge, lambda x: x.referential() and not x.adjunct):
            return self.next(self.edge, lambda x: x.referential() and not x.adjunct)
        return self.next(self.edge, lambda x: x.referential() and not x.find_me_elsewhere)
```

7.3.3 Complement selection

```
def complement_match(self, const):
    return const.check(self.licensed_complements())

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

def nonlicensed_complement(self):
    return self.proper_selected_complement() and
self.proper_selected_complement().check(self.complements_not_licensed())

def missing_mandatory_complement(self):
    return self.get_mandatory_comps() and (not self.proper_selected_complement() or not
self.proper_selected_complement().check(self.get_mandatory_comps()))

def complement_not_licensed(self):
    return self.proper_selected_complement() and not
self.proper_selected_complement().check(self.licensed_complements())

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

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

def does_not_accept_any_complements(self):
    return self.check({'-COMP:*'})
```

7.3.4 Complex selection, projection principle

Complex selection has to do with whether a constituent is in a thematic or nonthematic position. These functions need to be simplified, but the operation requires the use of a representative dataset.

```
def nonthematic(self):
    return self.container() and (self.container().EF() and self.container().finite()) or \
           (self.container().check_some({'-SPEC:*', '-SPEC:φ', '-SPEC:D'}) and self == next((const
for const in self.container().edge()), None))

def specifier_theta_role_assigner(self):
    return not self.EF() and \
           not (self.selector() and not self.selector().check({'ARG'})) and \
           self.check_some({'SPEC:φ', 'COMP:φ', '!SPEC:φ', '!COMP:φ'}) and not
self.max().container().check({'-SPEC:φ'})

def projection_principle(self):
    return self.projection_principle_applies() and not self.container_assigns_theta_role()

def projection_principle_applies(self):
    return self.referential() and self.max() and not self.max().find_me_elsewhere and
self.max().mother and not self.max().contains_features({'adjoinable', 'SEM:nonreferential'})

def container_assigns_theta_role(self):
    return self.max().container() and (self.selector() or (self.is_licensed_specifier() and
self.max().container().specifier_theta_role_assigner()))
```

7.4 Semantics (narrow_semantics.py)

7.4.1 Introduction

Semantic interpretation is implemented in the module narrow_semantics.py which bleeds the syntax-semantics interface (LF-interface). It begins by recursing through the whole structure and interpreting all lexical elements that have content understood by various semantic submodules (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 4.11.3), phi-conflict detection, tail feature interpretation, inventory projection (Section 4.11.1), operator binding (Section 4.11.5) and pragmatic processing (4.11.4). The general idea is that lexical features are diverged into different semantic subsystems for interpretation.

7.4.2 Projecting semantic inventories (semantic switchboard)

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

```
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.[33] 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.

7.4.3 Quantifiers-numerals-denotations module

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

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

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

[33] 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

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 ⟨idx, constituent printout, constituent, denotations⟩, so that the whole recursion will return a longer list containing all expressions that were provided with this information. This list, which is a simple auxiliary representation that plays no role in the theory, will then be used to build the assignments. Function create_all_denotations() will return a set of denotations (global inventory objects) that satisfy the criteria stored in the QND entry. For example, a pronoun *he* can only pick singular male objects, and so on.

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

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

7.4.4 Operator-variable interpretation (SEM_operators_variables.py)

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

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

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

```
def find_overt_scope(head, operator_feature):
    return next(({'Head': head, 'Scope': scope, 'Overt': True} for scope in
      head.working_memory_path() if {operator_feature, 'FIN'}.issubset(scope.features)),
      {'Head': head, 'Scope': None, 'Overt': False})
```

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

```
    def interpret_covert_scope(binding):
        if not binding['Scope'] and 'OVERT_SCOPE' not in binding['Head'].features:
            return next(({'Head': binding['Head'], 'Scope': scope, 'Overt': False}
                         for scope in binding['Head'].working_memory_path() if
                          {'T', 'FIN'}.issubset(scope.features) or
                          {'C', 'FIN'}.issubset(scope.features)),
                          {'Head': binding['Head'], 'Scope': None, 'Overt': False})
        return binding
```

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

7.4.5 Pragmatic pathway

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

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

```
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_lexica
l_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.[34]

Topic gradient calculations are nontrivial for several reasons. First, they must ignore some noncanonical word order changes, such as those created by Ā dependencies. An interrogative direct object pronoun that occurs at the beginning of the sentence should not be interpreted as the topic. Second, the more noncanonical the position is, the more prominent the topic/focus

[34] 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.

interpretation tends to be. This is especially clear in Finnish. Third, this language allows one the topicalize/focus several constituents (multi-topic/focus constructions) and to perform sentence-internal topicalization/focusing.

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

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

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.

7.4.6 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 recover_arguments(self, probe):
        return self.interpret_antecedent(probe, probe.get_antecedent())
```

The function finds possible antecedents while *interpret_antecedent* provides verbal feedback for the user on the basis of the former. The antecedent is determined by the following functions, where finite control takes care of special finite control in Finnish.

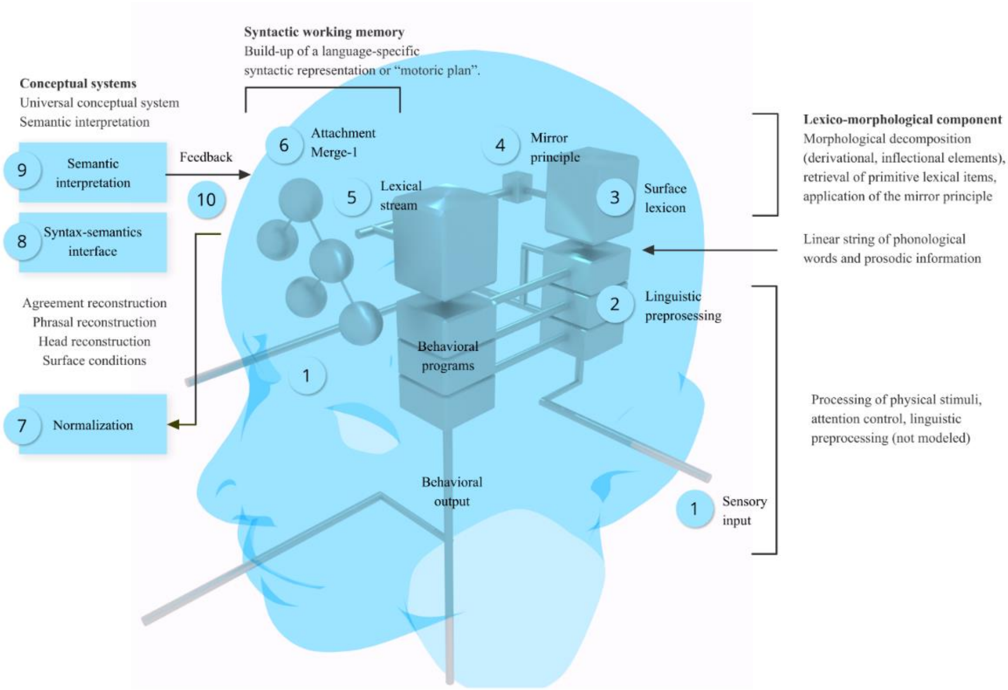
```
def get_antecedent(self):
    unvalued_phi = self.phi_needs_valuation()
    if {'PHI:NUM:_', 'PHI:PER:_'} & unvalued_phi:
        return self.control()
    elif {'PHI:DET:_'} & unvalued_phi:
        return self.finite_control()

def control(self):
    return next((x for x in [self.sister()] + list(takewhile(lambda x: 'SEM:external' not in
x.features, self.upward_path())) if self.is_possible_antecedent(x)), None)

def finite_control(self):
    return self.next(self.upward_path, lambda x: self.is_possible_antecedent(x) or
self.special_rule(x))
```

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.



Syntactic working memory
Build-up of a language-specific
syntactic representation or "motoric plan".
Conceptual systems
Universal conceptual system
Semantic interpretation
9 Semantic interpretation
Feedback
10
8 Syntax-semantics interface
Agreement reconstruction
Phrasal reconstruction
Head reconstruction
Surface conditions
7 Normalization
6 Attachment Merge-1
5 Lexical stream
4 Mirror principle
3 Surface lexicon
2 Linguistic preprocessing
1
Behavioral programs
Behavioral output
1 Sensory input
Lexico-morphological component
Morphological decomposition
(derivational, inflectional elements),
retrieval of primitive lexical items,
application of the mirror principle
Linear string of phonological
words and prosodic information
Processing of physical stimuli,
attention control, linguistic
preprocessing (not modeled)

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

8.1.2 Parse sentence (parse_sentence)

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

```
def parse_sentence(self, count, lst):
    self.sentence = lst
    self.start_time = process_time()
    self.initialize()
    self.plausibility_metrics.initialize()
    self.narrow_semantics.initialize()
    log_new_sentence(self, count, lst)
    self.parse_new_item(None, lst, 0)
```

8.1.3 Recursive parsing function (parse_new_item)

The recursive parsing function takes the currently constructed phrase structure α, a linearly ordered list of words, an index in the list of words and an inflection buffer as its arguments.

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.

```
def parse_new_item(self, ps, lst, index, inflection_buffer=None):
     if self.circuit_breaker(ps, lst, index):
          return
     [...]

def circuit_breaker(self, ps, lst, index):
    set_logging(True)
    if self.exit:
        return True
    if index == len(lst):
        self.complete_processing(ps)
        return True
    self.time_from_stimulus_onset = int(len(lst[index]) * 10)
    if not self.first_solution_found:
        self.resources['Total Time']['n'] += self.time_from_stimulus_onset
```

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

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

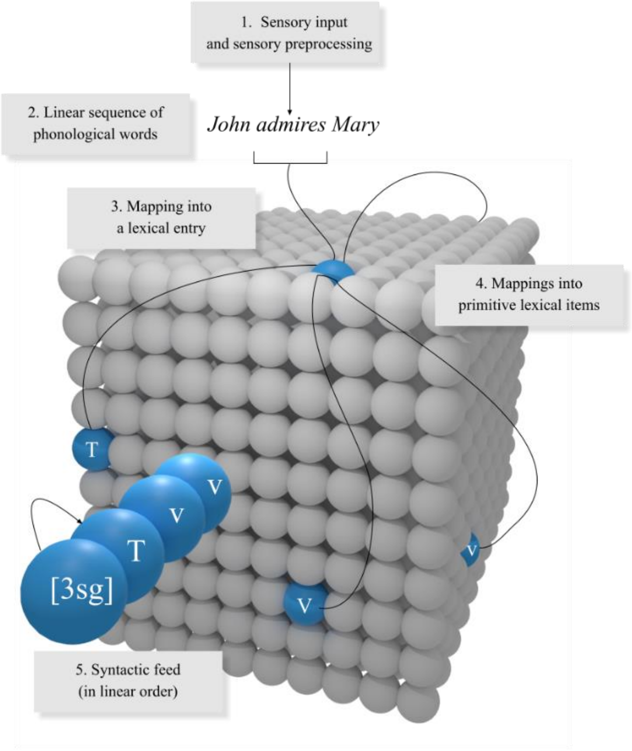


Figure 40. Organization of the lexicon. The lexicon is a mapping from phonological surface strings into constituents which make up the lexicon. Some surface strings map into primitive lexical items (blue items) which are sets of lexical features (e.g., /the/ ~ D). Other surface strings map into morphological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

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

Next an item from this list of possible lexical entries will be subjected to *morphological parsing*. Notice that a lexical item returned by the lexical retrieval may still consists of a morphological decomposition (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(lexical_constituent, lst.copy(), index)
```

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

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

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

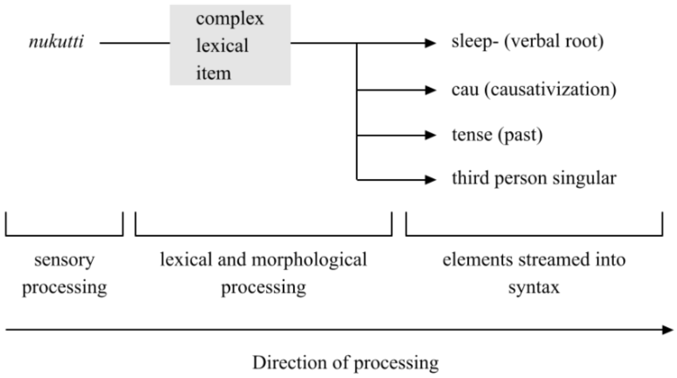


Figure 41. Lexical processing pipeline.

They will be merged to the existing phrase structure in the syntactic working memory, into some position. Merge sites that can be ruled out as impossible by using local information are filtered out. The remaining sites are ranked.

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

Each site from the ranking is then explored.

```
for site, transfer, address_label in merge_sites:
    new_constituent = self.attach(ps.target_left_branch(site), site, terminal_lexical_item,
     transfer)
    self.working_memory.remove_items(merge_sites)
    self.parse_new_item(new_constituent.top(), lst_branched, index + 1)
```

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

The code above contains two essential functions: *target_left_branch* and *attach*. 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 Merge-1 and sinking depending on the situation and performs working memory operations.

```
def attach(self, left_branch, site, terminal_lexical_item, transfer):
    self.working_memory.maintain(site)
    if left_branch.belong_to_same_word(site):
        new_constituent = left_branch.sink_into_complex_head(terminal_lexical_item)
    else:
        new_constituent = self.attach_into_phrase(left_branch, terminal_lexical_item, transfer)
    self.consume_resources("Merge", terminal_lexical_item)
    return new_constituent
```

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

```
 def complete_processing(self, ps):
     self.transfer.transfer_to_LF(ps)
     if self.postsyntactic_tests(ps):
         self.resources.update(PhraseStructure.resources)
         report_success(self, ps)
     else:
         self.narrow_semantics.reset_for_new_interpretation()
         report_failure(ps)
     if not self.first_solution_found:
         self.consume_resources("Garden Paths", ps)

def postsyntactic_tests(self, ps):
    return self.LF.LF_legibility_test(ps) and \
           self.LF.final_tail_check(ps) and \
           self.narrow_semantics.postsyntactic_semantic_interpretation(ps)
```

If these fail, the result is rejected; otherwise processing conteinues. 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 passes the filter are ranked. Ranking uses cognitive parsing heuristics, as explained below. The user can activate and inactivate these heuristics in the configuration file.

8.2.2 Filtering

Filtering is implemented by going through all available nodes (i.e. those which are in the active working memory) and by rejecting them if a condition is satisfied. When a parsing branch is closed by filtering, it can never be explored by backtracking. The filtering conditions are the following: (1) The bottom node can be rejected if it has the property that it does not allow any complementizers. (2) Node α can be rejected if it constitutes a bad left branch (left branch filter). (3) [α β] can be rejected if it would break a configuration presupposed in word formation. (4) [α β] can be rejected if it constitutes an impossible sequence.

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

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

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

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

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

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

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

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

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

If 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 ±15ms due to the way the operation system works. A simulation with 160 relatively basic grammatical sentences with the version of the program currently available resulted in 77ms mean processing time varying from <15ms to 265ms for sentences that exhibited no garden paths and 406ms for one sentence that involved 5 garden paths and hence severe difficulties in parsing.