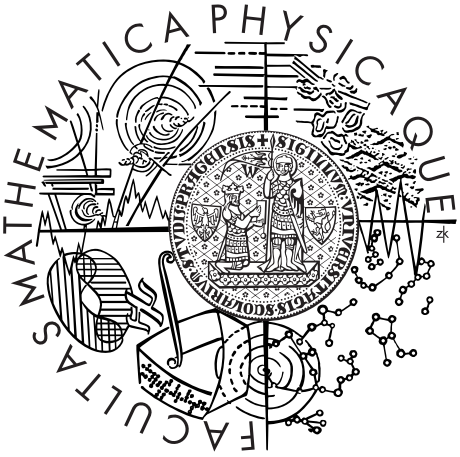
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Antonio Fernando García Sevilla

An online collaborative platform for the development of empirical grammars

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: Ing. Alexandr Rosen, Ph.D.

Study programme: Computer Science

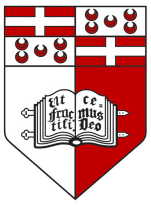
Specialization: Mathematical linguistics

Prague 2015

An online collaborative platform for the development of empirical grammars

Antonio Fernando García Sevilla

Faculty of Information and Communication Technology
Department of Intelligent Computer Systems



UNIVERSITY OF MALTA

September, 2015

Submitted in partial fulfilment of the requirements for the degree of
Master of Science in Human Language Science and Technology (HLST)

I would like to thank the LCT consortium, for designing the masters programme, and for the grant that made it possible for me to take part in it.

I owe my gratitude to Mike Rosner for his patience, and his efforts at guiding the chaotic creative process. To Alexandr Rosen for introducing me to HPSG, and giving wings to my forays into its use and implementation. To Markéta Lopatková, for her welcome into ÚFAL, a place where mathematics, computers and linguistics are one and the same. To Stan, an unexpected friend and ally in a foreign land. To my family, who were with me even when I wasn't there. And finally, to Benno, of like mind, and a best friend.

The work carried out for this master thesis resulted in a software product, a web application for online computational linguistics. It can be used and tested at the url `https://garciasevilla.com/bolde`, and the reader is encouraged to do so while reading this document.

Abstract

Modern science often requires the collaboration of large groups of people and the sharing of specialized data and knowledge. Computational linguistics is not an exception, with projects often involving researchers and data from different parts of the world and different languages.

In this document, a tool for internet collaboration in computational linguistics and its development are described. The tool is an online environment that enables users to manage and exploit different types of linguistic resources in a collaborative way. It features real-time editing of documents, visualization of complex objects and data pipelines for numerical and statistical analysis.

An extensive HPSG interpreter has been developed at the same time, and is also documented here. It has been embedded in the online tool, and a visual feature structure editor has been included. With their aid, a small grammar of Spanish has been developed, as showcase of the system capabilities.

One of these capabilities is the ability to extend the tool with arbitrary user code. In this document an example of this is provided, showing how the system might be used for interlingua or semantic translation.

Supervisors

Michael Rosner
Department of Intelligent Computer Systems, University of Malta

Alexandr Rosen
Institute of Formal and Applied Linguistics, Charles University in Prague

Contents

1 Introduction **1**

1.1 Wanted: an accessible system 1
1.2 Wanted: a collaborative environment 2
1.3 Wanted: a tweakable tool 3
1.4 Aims and Objectives 4
1.5 Proposed solution 5
1.6 Structure of the document 6

2 Background **7**

2.1 Grammar formalisms 7
2.1.1 CFG 8
2.1.2 Constraint-based grammars 9
2.1.3 HPSG 10
2.2 Implementations 11
2.2.1 NLTK 11
2.2.2 Trale 12
2.2.3 LKB 12
2.2.4 These are not the programs you are looking for 13
2.3 Internet collaboration 13
2.3.1 Document editing software 14
2.4 An empty niche 15

3 Design **16**

3.1 What it means to be a web application 16
3.2 Flux design 18
3.2.1 Graphical user interface 19
3.3 Engines 20
3.4 Why grammar formalisms 22
3.5 Borjes 23

4 BOLDE **25**
4.1 Client-server communication . . . 25
4.2 Server . . . 26
4.2.1 Overview . . . 26
4.2.2 Libraries . . . 26
4.2.3 Modules . . . 28
4.2.4 User files and configuration . . . 30
4.3 Client . . . 31
4.3.1 Compiling and serving . . . 32
4.3.2 Architecture . . . 33
4.3.3 Visual editor . . . 35
4.4 Engines and Pipelines . . . 37
4.5 Installation . . . 39
4.6 Structure of the code . . . 40
4.7 Summary . . . 40

5 Borjes **41**
5.1 Overview . . . 41
5.2 Why POJOs . . . 42
5.3 The type system . . . 43
5.3.1 Unification . . . 43
5.3.2 Unifiable types . . . 44
5.3.3 Non-unifiable types . . . 57
5.3.4 Parsers . . . 61
5.3.5 Trees . . . 62
5.4 Using the library . . . 63
5.4.1 Unify . . . 63
5.4.2 Copy . . . 64
5.4.3 Normalize . . . 64
5.4.4 Comparison . . . 64
5.4.5 YAML Grammars . . . 65
5.4.6 Packaging and install . . . 65
5.4.7 Structure of the code . . . 65
5.5 Borjes-react . . . 66
5.6 Summary . . . 69

6 Results **70**
6.1 Use case 1: A numerical, lexicalized CFG grammar 70
6.1.1 The base grammar 70
6.1.2 Lexicalized 74
6.1.3 Numerical 74
6.1.4 Pipeline configuration 76
6.1.5 Code extensions 76
6.1.6 Treebank 79
6.2 Use case 2: An example HPSG grammar of Spanish 80
6.2.1 The signature 81
6.2.2 The constraints 83
6.2.3 The lexicon 86
6.2.4 Parse trees 93
6.2.5 An extension: semantic trees 95
6.3 Testing 97

7 Conclusions and future work **100**
7.1 Conclusions 100
7.2 Future work 102
7.2.1 Evaluation 103

Chapter 1

Introduction

This document is the result of research work for a master thesis in Language and Communication Technologies. It records the design and implementation of a software system, which is online and working, and which strives to be a development platform for a kind of computational linguistics. But why should we need this? What is the motivation for such a project?

1.1 Wanted: an accessible system

Let us imagine Anna. She is a PhD student in linguistics, and is doing her dissertation on the aboriginal language of a remote population in the Amazon. She wants to document her discoveries in a formal way. She wants to produce a program that can translate basic health and legal information from the government for the tribespeople. She has many notes on the vocabulary and grammar of the language, and she thinks she has a pretty good grasp on it.

However, she is not an expert on any computational formalism. She is not a programmer, and she has only had a brief computer science course before. She has installed Python and NLTK on her computer, but they are programming tools, not scientific theory builders. She has tried to install LKB[1] and TRALE[1], but has never before used Linux. She does not understand what a live CD or a virtual machine is, and Emacs or Prolog are names more foreign to her than the aboriginal language she has been studying.

She decides to focus on the theory, and leave the implementation for later. She arms herself with a book on HPSG[1], a successful grammatical formalism, and tries to get to learn it. She draws impressive feature structures in pencil, but does not really understand what is going on. She is not sure whether what she is

[1]See chapter 2 for descriptions of LKB, TRALE and HPSG.

doing is correct, because she cannot test it. She can see it is similar to diagrams seen in articles, but she does not understand many of the design decisions. She cannot evaluate whether they are valid for her language either, since she cannot see what results the grammar she is writing yields.

What she would like to have is a visual tool, something where she could draw the structures of the formalism, and immediately test them on her data. That way she could progress incrementally in complexity, learning the insights and tricks of the formalism by seeing them in action, and understanding how they actually work. And she could draft her grammar in this system, and test and improve it, without having to write any code.

1.2 Wanted: a collaborative environment

Now let us imagine a different situation. A group of experts on the Spanish language wants to put together a formal grammar of international Spanish. The group comprises linguists from mainland Spain, in Europe, and from different countries in Latin America, as far apart as Mexico in the north and Argentina in the south.

On top of being long-hour flights away from each other, which makes frequent travelling expensive and impractical, some of them live in different timezones. They think they should be able to use the Internet. They would like to have a platform online, where the project could be discussed, and where the ongoing work could be stored.

There exist a number of software tools for this kind of collaborative work. They could decide on one of them, and with the aid of e-mail, solve communication. But this still leaves synchronization unsolved, the sharing of the ongoing work. How to each be able to work on their own, but still use and improve the same files?

There are some collaborative tools (e.g. Google Docs (*Google Docs* 2015), ShareLatex (*ShareLaTeX* 2015)) which offer simultaneous editing of files, but they are centered on text documents. The group could write the grammar as a plain text document, but this is not desirable. Consistency and exhaustiveness in the formal descriptions would mean extra work, since the underlying document software does not understand them. And turning the grammar into something a computer could use would not be straightforward, requiring constant translation to another format, which would also mean duplicating work.

Another possibility would be to use an approach more similar to that of a

software development team. They could write the grammar in one of the available formalisms (LKB, Trale) and put the code in a source control repository. After all, this is how many of such systems are developed. However, they are not programmers, and the installation and setup of this whole arrangement looks daunting. They would require on-site IT support, too, since they would also need the different programs in their local computers. And they would have to learn the programming language used in the chosen tool, not only the theoretical formalism.

What they wish they could have is a platform where they could write computational grammars, but that would also offer online features, including *collaboration* and *synchronization* of work.

1.3 Wanted: a tweakable tool

Our last imaginary scientist is John. He has a hypothesis on the behaviour of clitic pronouns in a particular language. His hypothesis is an empirical one, dealing with probabilities and some numerical calculations. He wants to test this hypothesis on a large corpus, and here lies the problem.

His theory is not a statistical, word-vector based algorithm. It requires succinct linguistic insights, of the kind most native speakers of the language would be able to produce, but which make computers struggle. He knows the phenomenon has been described in terms of a grammatical formalism, and it is indeed this description that he wants to extend with his mathematics.

He first tries to use a regular programming language, and its NLP libraries. He finds that he would have to program a whole formal grammar interpreter, just to get started, before being able to work on his hypothesis. He then turns to more sophisticated programs. He, like in the situations before, wants to use HPSG. He is able to install the tools already mentioned, like LKB and TRALE, and has worked with them before.

But they do not offer the functionality he needs. After all, what he is doing is a very novel approach to some linguistic phenomenon. He knows how to express his hypothesis as a simple algorithm. He would just need to encode the numerical computations in some kind of grammar rule, but the framework does not expect this. He would need to dig into the implementation, for which he would first need to understand it, when he is more of a theoretical person.

If only he could just write some simple function code, and tell the system to use it when parsing the pertinent clitics.

1.4 Aims and Objectives

As we have seen, existing tools do not provide full support for our hypothetical scientists' requirements. Some of the programs offer partial solutions. Some of them offer the user some power and flexibility, but either are too low level, or installing and using them requires a great deal of mastery and the management of very complex setups.

The intention of this master thesis is to build an example solution to these problems. Due to time constraints, it does not try to build a complete system, but rather a demonstration of how such a system would work and might be implemented. It also provides a reasonable basis, a foundation upon which a more complete solution could be built.

Based on the examples, which are just dramatizations of real user experiences, the following aims will guide the development:

- **Linguistically oriented**. The system should support computational linguistic work, and offer NLP capabilities.
- **Accessible**. The system should require the minimum setup and training possible, ideally zero.
- **Intuitive**. Interaction with it should be visual, and as close as possible to the theory instead of the implementation.
- **Collaborative**. Users must be able to share work on the system, and development should be interactive and collaborative.
- **Open**. Hooks for user extensibility should be provided, and the system should be flexible and future-proof enough to support different computational linguistic paradigms and experiments.

These aims can be refined and complemented by the following, more concrete objectives:

- **Cross-platform**. The system should work on Windows, Linux, Mac OS, iOS, and Android. All functionalities should perform sufficiently well in all of them.
- **Open source**. All software used for the system, and the system itself, should be freely accessible by any interested person, both for use and installation.

- **Live editing**. Users should be able to edit files simultaneously. This means that a change made by a user should be seen by another user editing the same file as quickly as possible in order to prevent conflict.
- **Persistent**. User work on the system should be able to be saved. Ideally, the saving should be performed automatically, to prevent data loss.
- **Visual**. The system should support a graphical user interface, and results and grammatical objects should be able to display visually.
- **Powerful**. The system should offer enough power to work with a sufficiently complex computational linguistics paradigm. Context-free, unification based grammars make a specialized but broad enough target. As many as possible of the characteristics of these grammars should be exploitable in the system.
- **Empirical**. Mathematical and algorithmical constructs should be able to be used in as many as possible linguistic components, ideally all.
- **Configurable execution**. The steps taken by the system to test or execute a linguistic project should be configurable by the user.
- **Extensible**. Extensions in the form of code must be supported by the system, in order to customize or improve its behaviour.
- **Interoperable**. Results from the system should be usable in other implementations, and resources for these should be able to be imported into the system.
- **Multilingual**. Support for different languages should be provided, and not be restricted to any particular character set or language class.

1.5 Proposed solution

The solution is, in essence, a web application. It is a program which a user accesses over the Internet, through a web browser. It is a development environment, which allows users to create projects and edit files. These files can then be tied together in an execution pipeline that allows them to be run. The execution is user configurable, but the system also offers a native linguistic interpretation.

And at the same time, all actions are amenable to collaboration. Users can edit files simultaneously, live, and can communicate with each other about the project

status. Their work is saved automatically, and is immediately accessible from anywhere in the world. All of this without the need to download any additional software in their computers, apart from a web browser, and with no need for installation or system setup.

1.6 Structure of the document

Before diving into the software solution, some background must be provided. This is done in chapter 2, which explores a few existing tools, and explains the theoretical paradigm of formal grammars, which will be the basis for the linguistic engine.

Chapter 3 details the design of the system. It talks about the different components, and how they will have to be tied together. Chapters 4 and 5 then go deep into the implementation, documenting the software development process on which most of the work has been spent.

Chapter 6 evaluates the aims and objectives achieved. It exemplifies how a fictional user, like those from the beginning of this chapter, might use the system. Chapter 7 wraps up the document, with the conclusions and a look at the future work to be done.

Chapter 2

Background

Linguistics is often considered to be a part of the humanities. To some, this means theories and statements difficult to computerize, not ruled by the strict and hard logic of mathematics and other parts of science. However, modern linguistics is indeed a scientific discipline. One particular branch of linguistics has been very logical and even mathematical from the very beginning, making it ideal for its use in computer science: the branch of grammar formalisms.

2.1 Grammar formalisms

Ever since Chomsky stated his original hierarchy of languages (Chomsky 1959), different paradigms have emerged to try to improve the way in which a computer may make sense of language, a human capacity. It is a matter of discussion whether natural language is indeed a formal language, but this hypothesis has seen the most advances in both theory and implementation. Thanks to these paradigms being so intent on formalism, they are easily computerized, and have often been.

In mathematical terms, a formal language is nothing more than a set of strings, and its formal grammar is a description of these strings. For a trivial, finite language, an enumeration of all the strings might seem enough. However, most interesting languages are infinite, so that an enumeration of the valid strings is not possible. Moreover, a mere listing of words does not give any information apart from whether a string belongs to a language or not.

Formal grammars can be seen as algorithms, ways of determining the strings in a language without having to list them all. On top of being a much more efficient way to describe a language, formal grammars also have a structure, which reflects the internal structure of the language itself. Thus, they are also

powerful ways of understanding it, finding its internal arrangement, and how the different components relate to each other.

By using a formal grammar to describe a string, a process called parsing, a formal description in terms of the grammar is obtained. This description can then be used for understanding what the meaning of the string is, and how it relates to other strings in the language. In the case of natural languages, being able to extract the meaning of a sentence is the ultimate goal, a process that we seem to perform in our brains almost unconsciously but which is still far from solved for a computer.

In order to achieve this goal, many grammar formalisms have been and are still devised. A grammar formalism is a theory on how to construct formal grammars, a description of their power and structure. From another, compatible point of view, it is a theory on the structure of language, which then immediately translates into a way to build formal grammars.

2.1.1 CFG

For many classes of languages, the most useful grammars are sets of rules which transform symbols repeatedly, to either convert the text into its formal description (parsing) or generate a string given its description (generation).

Context-free grammars are one such class of grammars. They are generative grammars, in the sense that they are fully explicit, covering the whole of language. They describe strings in the language by giving rules for obtaining them from an internal representation. In the CFG formalism, this representation is a single symbol `S` (for sentence). The CFG rules transform `S` into other symbols (non-terminals), which then are transformed again, repeatedly, until they generate all the strings in the language.

For the inverse process, that of parsing, the characters in the "surface" string (the word or sentence) are transformed, by combining them into the non-terminal symbols. These are again combined with each other iteratively, according to the rules, until finally the single symbol `S` is reached. If this can not be done, the string does not belong to the language.

The important characteristic of context-free grammars is that the rules that transform a non-terminal into other non-terminals do not depend on the context, that is, the adjacent symbols in the string, or any external information. The process of applying the rules iteratively gives rise to a chain, the set of steps used. This chain forms a tree, called the parse tree or derivational tree, with the symbol `S` as the root node, and the characters in the string (or the words in the

sentence) as the leaves, or terminal nodes. It is this tree that gives information about the structure of the sentence, and the nodes and their relations encode its meaning.

Many languages are context-free, meaning that they can be fully described by a context-free grammar. For example, the source code of programming languages is almost always parsed with this kind of grammars, and each node describes an operation which the computer must perform.

However, human language is often thought not to be context-free (Higginbotham 1987), (Shieber 1987). There are constructs in some languages that are provably not describable by a CFG, which might make us think that it is not a sufficient formalism. Nonetheless, most phenomena of natural language are indeed context-free, and many principles are still valid (Pullum and Gazdar 1982). And while basic simple CFGs might not be sufficient to describe human languages, the formalism can be extended to make it more powerful and expressive.

2.1.2 Constraint-based grammars

One such extension comes in the form of constraint-based grammars (Shieber 1992). These grammars extend the symbols used in the internal description of a language (the non-terminals), and instead use complex representations, which encode structured information. For example, instead of using just a symbol `NP` for a noun phrase, a constraint-based grammar might use a feature structure, a multidimensional description of the phrase. The feature structure can be seen as a vector of features, each holding an appropriate value that encodes some linguistic information. And it can be recursive, meaning that the value of a given feature can again be a feature vector, allowing for hierarchical organization of the information.

Rules in constraint-based grammars often generate very general nodes, or too many possibilities, and here is where constraints come into play. Constraints express different linguistic insights that restrict the possible shape of feature structures. They state the relations of the features in a node with one another, how they might combine and how they may not. This narrows down the possibilities from application of the rules, ending up only with those that encode the actual strings in the language.

The paradigm of constraint-based grammars allows for encoding information that context free rules might be unable to handle, or cannot do it in an elegant manner. For example, unbounded constructions, which are common in human languages, are phenomena where a word is related to another word in the sentence

which might not only not be adjacent, but can actually be arbitrarily distant. Constraint-based grammars allow for dealing with this phenomena in a way that is intuitive and efficient, by encoding extra information in the nodes.

2.1.3 HPSG

HPSG, Head-driven Phrase Structure Grammar (Pollard and I. Sag 1994) is one good example of a constraint-based grammar formalism. For its internal representation of grammar objects, it uses feature structures, which encode all manners of syntactic and semantic information. It is a monolithical paradigm, in the sense that all levels of language (morphology, syntax, etc.) are handled with the same mechanisms, and every feature structure encodes all available linguistic information, of all levels, pertaining to the corresponding node.

One of the characteristics of HPSG grammars is that phrase structure rules are driven by the head features of the nodes. This feature encodes the syntactic information of the head of a phrase, and is one of the main restrictions when combining subtrees to form phrases.

This information, and much more, is actually kept in the feature structure description of a word. HPSG is a heavily lexicalized formalism, meaning that as much information as possible is stored with the words of the language. Thus, words not only have features such as gender, number or case, but also valencies, which express what other words they combine with. In HPSG, most of the linguistic knowledge needed for combining a set of words into a sentence belongs to the words themselves.

Apart from the feature structures for words, HPSG uses constraints, which state how these structures can be put together to form phrases. Apart from the phrase structure rules, which are equivalent to the rules in CFGs, HPSG has additional principles. These principles restrict the shape of the feature structures that result from the application of the rules, using the information received from the words to determine what combinations of them are valid.

HPSG grammars also have another constraint, and that is that feature structures belong to a given type (Carpenter 2005). This type restricts what features they can have, and what types of values the features can have. Types are organized in a hierarchy, the signature, which also allows for some types to specialize others. This way, a feature structure can be of a general type, meaning it has a given set of features, their values determined by the linguistic entity it represents. Later it can be constrained to be of a more specific type, a subtype in the signature, which gives more information on the kind of entity it represents, by

restricting the possible entities to the ones represented by the subtype.

Apart from being very expressive from a linguistic point of view, HPSG is a theory inspired by computer science too. Its way of representing data in feature structures maps very well to computer programs, and its rules and constraints can be encoded as programming constructions directly from the theoretical paradigm. Therefore, there are a number of HPSG implementations, and they have been successful in translating a theoretical formalism for describing language into actual programs that understand it according to the theory.

2.2 Implementations

As we have repeatedly mentioned from the start of the chapter, the driving characteristic of this paradigm of linguistics is its ability to be implemented as computer programs, so it is expected that different software already exists to do so.

2.2.1 NLTK

One very well-known platform for doing NLP is NLTK (*NLTK – Natural Language Toolkit* 2015), a python library for computational linguistics. It gives easy access to many existing resources, and has modules for dealing with various aspect of NLP and many different algorithms implemented.

It has a very nice landing page, and a book for getting started with it (Bird, Klein, and Loper 2009). However, the documentation is patchy, and some modules have very little information on how to make use of them. It has CFG and PCFG modules (an extension to CFG with probabilities), a dependency grammar module, and even a feature structure grammar one.

In addition, NLTK is just a programming library, a toolkit. It requires the user to learn and understand Python, a computer programming language. It does not provide any ready-made solutions, but rather allows the user to build (by programming) what they need, offering functionality in the form of exported modules.

For a user without some background in programming, learning to program, and learning how to use NLTK, is an additional burden. On top of it, NLTK will also require the user to build a computer program for using their results, having to deal with engineering details like input and output, graphical interfaces, and data storage.

2.2.2 Trale

An existing application geared for computational linguistics is TRALE (Penn and Haji-Abdolhosseini 2003). Its focus is on formal grammars, in the form of HPSG. TRALE is a development environment, based on the attribute logic engine ALE. ALE is written in Prolog, and features many of the characteristics needed for implementing an HPSG grammar. It integrates phrase structure parsing, semantic generation, and constraints application on feature structures. TRALE is the user software on top of ALE, allowing users faster development and visualization of results. TRALE can display feature structures and derivation trees resulting from the parse, with many options for inspecting them.

To use TRALE, one should have an appropriate computer environment. Linux is a must, and a number of supporting libraries have to be installed (a Prolog interpreter, the Grisu user interface library, and Graphviz for viewing the signatures, among others). Distribution is made in the form of tarballs, a common format in the Linux community. An alternative to this installation is provided, by downloading a live CD, with a customized Linux distribution for using TRALE: Grammix. This distribution has a Linux system, with a graphical desktop environment, TRALE and required libraries already installed, and even a set of example grammars included.

While the Grammix live CD is great for introduction, it forces the user to work in a foreign environment, and restart the computer just to work on it. Installation on a virtual machine is hardly better, requiring some technical knowledge and being often painfully slow. It is better to install TRALE directly on Linux, and use it as a locally installed program. However, documentation is very scarce so to make good use of TRALE, a user must be familiar enough with both development on Linux and the HPSG theory. The computational paradigm it relies on, logical programming in Prolog, is also not very fast, and not very well-known among today's developer community.

2.2.3 LKB

Another development environment for computational linguistics is LKB, the Linguistic Knowledge Builder (Copestake 2002). It is a framework for developing grammars and lexicons, contained in the general paradigm of unification-based formalisms. It is not restricted to HPSG, but it is mostly used for it. It is tightly related to DELPH-IN (DEep Linguistic Processing with HPSG – INitiative[1]), and

[1] `http://www.delph-in.net/wiki/index.php/Home`

implements its reference theory. Other DELPH-IN tools integrate comfortably with it, and provide pre-processors and helpful tools for grammar development.

LKB is implemented in Common Lisp, and it generally follows a development philosophy that is quite common in some artificial intelligence circles. Lisp as the language and Emacs as the text editor, while not a requirement, make a powerful combo. It can provide a user interface based on Motif, which is a somewhat obsolescent GUI library.

As was the case with TRALE, LKB is best used in Linux. Some Microsoft Windows builds exist of the software, but they are old, and no newer versions can be built because of licensing issues. Mac OS X, which is closer to UNIX architectures and thus easier to make compatible, has some, but limited support. Again as with TRALE, a Live CD is provided for running LKB without having to install it, with the same set of penalties that were discussed before.

2.2.4 These are not the programs you are looking for

As with TRALE, using LKB requires a quite steep learning curve, and adapting to user interface and programming paradigms which are not so common nowadays. The effort of getting started is non-trivial, and can put off a potential user who is undecided on whether to dive into the formalism or not. Ease of use is not the most salient feature of computational linguistics tools. On top of it, no collaborative features are provided by the systems. Installation and development are local, and if the user wants to share their work and synchronize with a larger group, another whole setup has to be built and maintained.

2.3 Internet collaboration

The Internet was invented for collaboration, as an advanced communication method for military command and operations. The Web was then created, and e-mail was born. It has become a primary tool for collaboration, combining the structure of written communication with the speed and immediacy of live conversation.

As Internet infrastructure has improved, more and more applications have been built on it. Nowadays, bandwidth and latency are exponentially better than just twenty or thirty years ago. The programs used for accessing the Web have also evolved, becoming powerful platforms that can house, and where indeed are housed, many fundamental applications for today's ever connected life. Bank accounts can be managed via the Internet, companies have employees around

the world who do not meet in person regularly, but are still synchronized to the second and up-to-date with the tasks at hand and the state of the project.

Some of the modern, web applications that have come into existence in recent years are focused on a more specific type of collaboration, that of developing documents or other type of resources in a group. They allow their users to work together, at the same time, on the same piece of work, even when physically miles away.

2.3.1 Document editing software

An example of this is ShareLatex (*ShareLaTeX* 2015). ShareLatex is a document editing application, which provides the user with an interface for writing Latex documents on the web. Changes are automatically saved to a back-end server, and documents can be compiled on demand, visually appreciating the result of the editing process in real-time. And, as its name suggests, it lets users share projects, and collaborate on them. It goes further, allowing editors to write and make changes simultaneously to the same document. That way, a group of people can develop a document by actually writing it at the same time, helping each other and giving suggestions, or dynamically distributing work without having to stop and synchronize every so often.

Another example is Google Docs (*Google Docs* 2015). Part of a complete office suite on the web, Google Docs offers applications for creating spreadsheets, presentations, or writing rich text documents. The document editor is WYSIWYG (What You See Is What You Get), meaning that the editing process is performed on top of the actual finished result, and the final appearance of the document can be seen while working on it. And Google Docs also allows live editing, simultaneous communication, in this document software.

These applications are good solutions to some of our requirements. They allow users to collaborate in real time, they automatically save changes, and are straightforward and intuitive. They have already solved the problem of synchronizing their users, and are still evolving and improving in the features they offer. However, they do not, and certainly will not, support projects in computational linguistics. They are designed to help human users write text documents, which are then read by other human users. They do not pretend to offer a solution for creating documents which need to be more than just text, more than just presentation, but rather have a defined meaning understandable by a computer.

2.4 An empty niche

We have seen that there exist computational linguistic formalisms that serve to instruct a computer on how to handle human language. We have seen that there are software applications for developing them, and for translating the human-made theory into computer-understandable algorithms. There are also libraries and programming toolkits for doing computational linguistics in a data-oriented way, and for programming custom solutions to NLP problems. Finally, we have also seen software for editing documents in a collaborative fashion, with simultaneous work by groups of users supported.

These all cover our requirements from chapter 1, however, they do not all do so at the same time. In the intersection of our requirements there is a void, an empty space for a program that can do formal linguistics, and computational NLP applications, and all of it in a collaborative way, with large groups of developers joining efforts and working together on a solution to their problem. Since such an application does not seem to exist, this master thesis was dedicated to building it. In the next chapter, its design is discussed, and its architecture and functionality explained.

Chapter 3

Design

The objectives laid out in chapter 1 can be thought of as a list of requisites for our software solution. The aims of instant set up and universal access lead us to a web application. Once this decision is made, collaboration comes almost free. Web applications, which make use of the Internet infrastructure, are ideal for sharing work and collaborating in real time.

User friendliness is a question of making the interface think of the user, rather than as a mere extension of the underlying system. The interface shall allow digging deep into its capabilities, but also be intuitive and straightforward to get started with. A graphical interface, in the form of a browser application, meets these expectations. It offers extensive flexibility and graphical power, and users are already used to its use experience and metaphors.

What is left is to make the system open enough, clearly architectured, so that other experts and paradigms can participate in the system features. Modularity, and clear separation of concerns in the software design help us achieve these aims.

In this chapter, the design of our software solution, code-named BOLDE[1], is presented.

3.1 What it means to be a web application

Every paradigm choice opens up a series of possibilities, but also comes with a set of restrictions. A web application is extremely portable, and distribution is trivial. Basically, the program runtime is provided by the system vendors in the form of web browsers, and the distribution infrastructure is a most pervasive one: the Internet. Of course, this also imposes tight constraints on the technologies that can be used.

[1]See chapter 4 for the meaning of the name, and the system implementation.

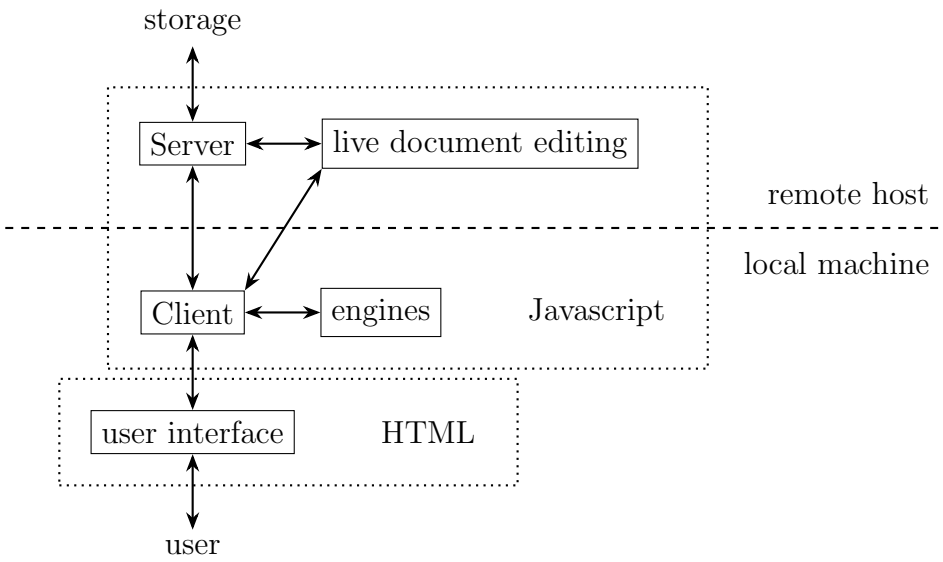


Figure 3.1: Global architecture

A basic consideration is that of the split of the program, between the remote host where the application is stored (the server), and the local device, the user's machine which functions as the interface (the client). As in other traditional client-server architectures, a number of options exist for web applications.

Thin client designs give most responsibility to the server. All the work is done there, and the client merely acts as a rendering surface for the interface. A thin server design, on the other hand, makes the clients do most of the work, and the server merely serves as a store for the program code. BOLDE takes a hybrid approach.

Most of the work is done in the client, along with the interface and user input. The server does have some responsibilities, though, apart from serving the program assets. It serves as synchronization point, storing user files and projects, and it is the middleman organizing communication between clients. Such an architecture is very common today, in what is often called "the Cloud". A very thorough overview of this paradigm is given in Armbrust et al. (2010).

Another constraint that web applications must face is that of the technology stack of choice. Server code is somewhat more free, any program that can run in the server machine is appropriate. But for the client code, a more restricted set of options exist.

A recent development is that of HTML5, which –apart from the HTML language version 5 proper– is often used as an umbrella term to also cover Javascript (technically called ECMAscript, versions 5 or 6) and CSS version 3 (Anthes 2012).

These are all languages standardized by international bodies, with open source specifications and implemented in modern browsers. One of the advantages of this stack is that it also extends to the server.

Node.js (*NodeJS* 2015) is a server runtime that also uses javascript as its native language. Using the same technologies throughout the whole of BOLDE improves the speed of development and maintainability. This practice is also a modern and upcoming paradigm, and so it is most appropriate for developing a web application such as BOLDE. An overview of this global design can be seen in Figure 3.1. The reasons why some of the components are drawn separately will become clear when the implementation is discussed, in the following two chapters.

3.2 Flux design

As per one of our global aims, the code for our software system has to be modular. But we have also decided on a heavy client, which retains many capabilities and does not delegate every operation to the server. This makes designing the interface architecture a delicate issue, that must be chosen right from the beginning. For BOLDE, the FLUX[2] design will be used.

In older web applications, the functionality was divided in pages. Every operation required a call to the server, which rendered a different view (html page) in response. Application state was implicit in the current path and the browsing history, or had to be stored by the server or in cookies. Modern applications are single-page, meaning that the user does not have to refresh the current page, or browse through a site map. This helps by retaining the whole of the application state and user interactions in the same place, but makes modularity and design complicated.

Flux is an architecture designed precisely for single-page applications. It subdivides the functionality of the user interface, such as our BOLDE client, into three main components: a dispatcher, a set of stores, and the views. Data flow in the system is unidirectional. Each component has a defined responsibility, receives data from only one other component, and produces data for only one other component. They are arranged in a cycle that closes data flow, as can be seen in Figure 3.2.

The dispatcher starts the data flow, signifying that an action has been taken (external action in the diagram). The stores, which keep the application state, listen to the dispatcher, and update themselves as appropriate for the actions

[2] `http://facebook.github.io/flux/`

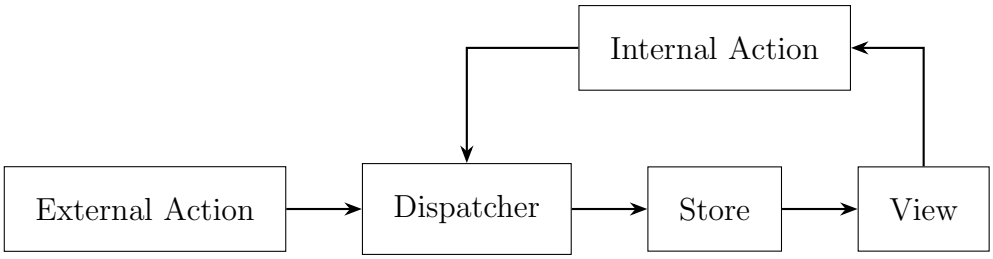


Figure 3.2: Flux architecture diagram

taken. Finally, the views render the state into the screen, by requiring information from the stores. They also offer the user means of interacting with the system, and these interactions (button presses, clicks, edits) are interpreted as actions (labeled internal in the diagram) to be sent to the dispatcher. Thus the cycle ends, and the whole flow is enabled, while keeping dependencies to the minimum, each subsystem only depending on the previous one.

3.2.1 Graphical user interface

An important part of the flux cycle are the views, which make up the graphical user interface. The GUI has to expose all the functionality of the client, both informing the user of the application state and enabling them to change it. In BOLDE, the main objects the user interacts with are files. The user interface should allow files to be displayed and edited.

Files could be displayed one at a time, making use of the full screen. But this would not allow for more than one file being displayed at the same time, for comparison or simultaneous editing. Ideally, the application should be able to display as many files as the user requires. This would quickly become unmanageable, so a compromise can be reached. Like many Integrated Development Environments, such as Visual Studio[3] or Eclipse[4] do, the screen can be divided into three regions, and each can display a different file. The regions can be resized, letting the user configure their desired workspace.

If more than three files are open, not all of them can be displayed at the same time. A common user experience solution is that of tab panels. Each panel can have different tabs, displayed for example in a bar at the top of the region. The rest of the region will show one file at a time, and changing what file can be displayed is achieved by clicking on the tabs. To make the interface even more

[3] `https://www.visualstudio.com/`
[4] `http://www.eclipse.org/`

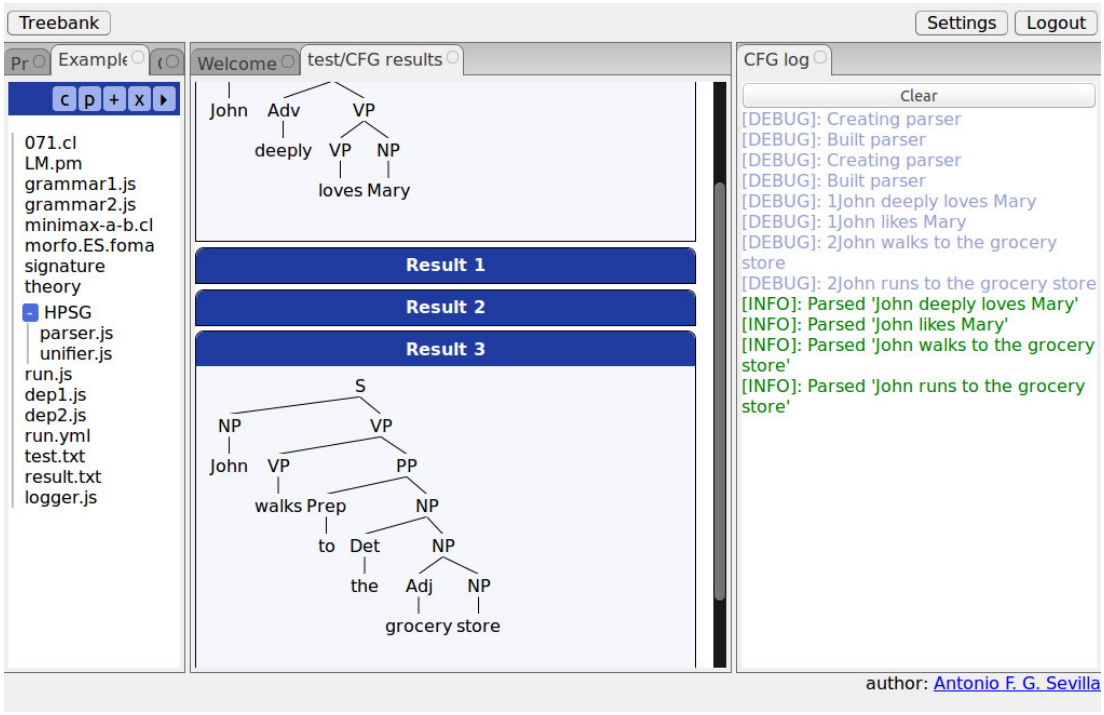


Figure 3.3: Example of regions with results

configurable, the user can move tabs from region to region.

A final advantage to the tab approach is that every other interaction with the system can be displayed in this paradigm. For example, another important object of BOLDE is that of projects which organize user files into logical units. The project list for a user can be displayed as a tab, and the file list for each project can, too. The actions for each project can be a toolbar on the top of the region, with buttons for the different interactions.

Other types of information that the system might want to display can be put in the form of files, too, and collected in a tab. A log of system information, user messages, or a list of parse trees from a pipeline execution; all of these can be made into a "mini page" which then can be displayed in a region, and configured via the tab mechanism. Some examples of what has been discussed in this section can be seen in figures 3.3 and 3.4.

3.3 Engines

Now that we have the user interface, the user can log into the system, and edit and store files. The server is in charge of providing support for these operations, and also synchronizes them among different clients. The only thing missing is actually using the files and resources that have been developed.

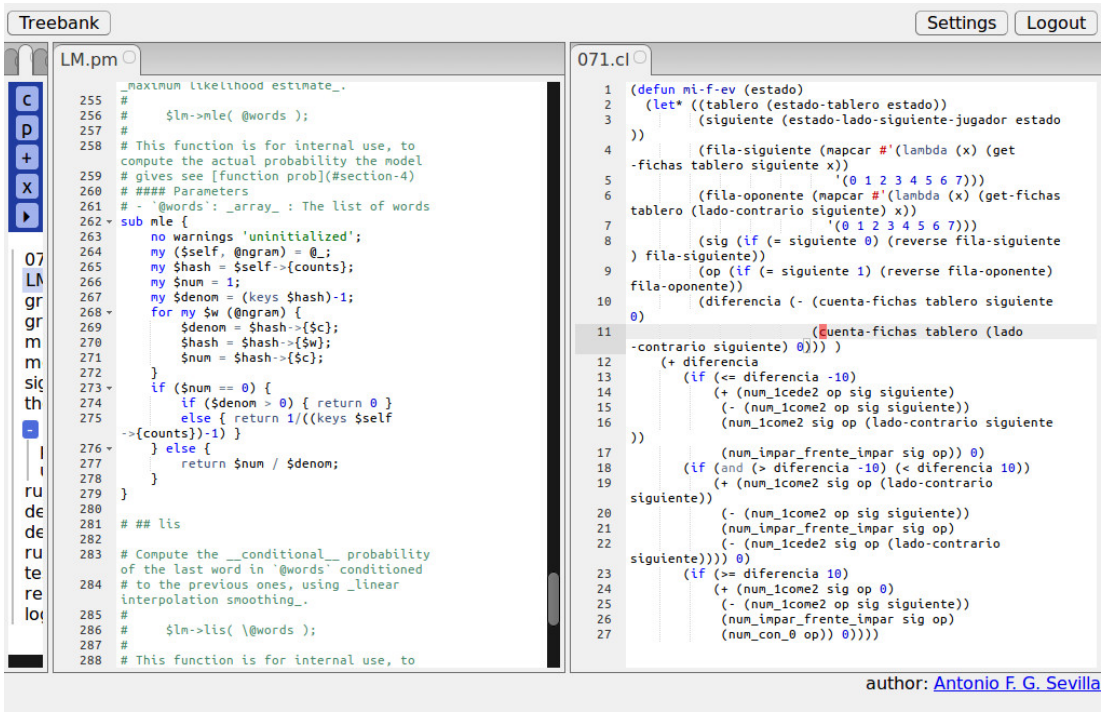


Figure 3.4: Example of simultaneously editing two files

For this, different runtime engines can be implemented. These engines are mini-programs which have a single responsibility. They can be seen as black boxes, and can be combined by the user into execution pipelines. The black boxes take some input, do some work with it, and optionally output some results. These can then be fed to another component in the pipeline for further processing. But these black boxes need to be executed by an interpreter somewhere.

There can be two solutions to this. One option would be to make the server useful also as a backend for executing the pipelines. Once the user has completed an iteration cycle, it asks the server for the execution results. The pipeline is run in the backend, and its products are then sent to the client for evaluation.

Another option, the one we choose, is to have the execution performed at the client side. The client has all files needed for this, and we can thus utilize their machine's resources for performing potentially expensive computations.

There are two advantages to this approach. On the one hand, it makes the system more scalable. The server does not have to support this additional load, and as the number of users increase, so do the available resources. Each user brings a machine, available for doing the required tasks. On the other hand, if the pipeline runs in the client, it can then be halted and debugged halfway through execution, a big aid for development.

There are also two disadvantages, though. This approach imposes higher re-

quirements on the device used to access the application, since it needs to support the underlying execution engine too. It also makes execution results not shareable, being as they are stored in the local computer of one user.

Nonetheless, there are easy solutions to these two problems. Both approaches are not exclusive, making server execution an option for less powerful devices. The engines can be written in javascript, the common language to server and client, and so they can be executed in both without modification. The other disadvantage, that of local results, is easily solved by having execution results also recordable to a shareable file, sent to the server and synchronized with all clients.

BOLDE will provide three main engines: a javascript interpreter, for executing arbitrary code (meeting the requirement for extensibility); a treebanking component, and a linguistic engine (for the requirement of computational linguistics as a first class citizen in the system). The linguistic engine will be a formal grammar parser and interpreter.

3.4 Why grammar formalisms

There are many ways in which computational linguistics can be done. As in other branches of artificial intelligence, natural language processing can be tackled from two different points of view (Tsujii 2011). In the data-driven approach, vast amounts of data are collected, and learning algorithms (often statistical ones) are used to make the system extract knowledge from the data.

The researcher can seed the system with some pre-existing knowledge, and the constraints on the algorithm can follow theoretical results on the domain. However, the main procedure is to extract the information from the data, to require as little human intervention as possible. This makes processing all the available data tractable in terms of both time and other resources.

The other side, the knowledge-based approach, takes the opposite point of view. Its goal is to encode expert knowledge in a form that a computer can understand, and to develop algorithms and tools that translate the theoretical results of the domain into programs and applications that can use them.

The first approach, the "empirical" one, does not necessarily require human expertise in the knowledge domain at hand, but rather benefits from expertise in software development and mathematics. The people who have this skill are often well versed on programming tools or statistical packages, that they can utilize to encode their vision. We are not trying to build a system for them.

It is the other field that our system should try to target. Experts in linguistics, who have theoretical models and results on language, but who maybe can not easily exploit the tools available to computer engineers and statisticians.

In this expert knowledge domain, one of the most successful paradigms is that of formal grammars. Formal grammars express the way in which text, the surface representation of language, should be interpreted (parsed) to understand its structure (and ideally, its meaning). Thus, formal grammars are the ideal tool for the theoretical linguist who also wants to see their results apply in computation to real data.

Simple formal grammars, like CFG, are implemented in many different software packages. On the other hand, some of the most complex and modern ones do not have an actual implementation. Somewhere in the middle stands HPSG. It is a very expressive and flexible formalism, but is also from the start very computer-oriented, and intent on the possibility of implementation.

3.5 Borjes

In order to build a linguistic environment, there must be a linguistic engine underneath. This engine takes care of the inner workings of the system, providing the functions for building and using the objects that the environment is centered on.

In our case, since BOLDE is an application for formal grammar development and parsing, there must be a component that understands and works with formal grammars. Making it a separate module is a good development practice, and also increases the possible future usefulness of the whole work, allowing potential users to only include the desired components.

There are already a number of implementations of grammatical formalisms in existence, like LKB (subsection 2.2.3) and TRALE (subsection 2.2.2). In general, these applications require local installation and environment setup. Since we want an online solution, a valid approach would be to install those applications remotely, and to design a web front end for them.

This is not ideal for a number of reasons. Having the front end and back end work in different languages, and even paradigms, makes the interface between them brittle and difficult to implement. Also, the final product would not be a single entity then, even with different components, but rather an incomplete system, dependent on other independent systems for its proper functioning.

Having the engine necessarily work in the backend would also introduce a

single point of failure. It would increase stress on the server, while not utilizing the resources available in the user's computer. But the user should be relieved from the need to install anything, or to deal with complicated setups. This discards most software stacks available. Compiled languages would require the user to compile the source, or the developers to provide compiled versions for all the different possible platforms.

Interpreted languages make the promise of letting the developer produce one single piece of work, which is distributed and which can be run without any hassle. In modern environments, however, this is seldom true. Java (Gosling 2000) has long been viewed as the most successful contender in this enterprise. Plagued with security issues, with an uncertain status as to its openness as a platform, it also requires the user to download and install the Java Virtual Machine. Programs written in Java then use their own interfaces and metaphors, offering a user experience completely independent from the rest of the user's system.

In the last few years, a platform has been growing that provides the interoperability, integration, and flexibility that we demanded. Web browsers have turned into truly self-contained computational platforms. There are operating systems centered on just a web browser (Pichai and Upson 2009). This platform, the browser, is shared by almost every user, regardless of operating system and choice of vendor. Its language is Javascript, which has evolved into a development stack, complete with dialects, compilers, and development tools; a stack that delivers just the connectivity and no installation requirements that we have.

Of course, being a relatively young computing paradigm, there is no such thing as a formal grammar interpreter written in Javascript. It was thus decided to write a custom one, which would serve as the engine for the BOLDE application. It was also decided that it should be an independent library, so the work done in it might be reusable in the future. This library is Borjes, which will be described in chapter 5. But first, in chapter 4, the implementation of BOLDE will be discussed, and how the design decisions that have been discussed in this chapter translate into an actual software solution.

Chapter 4

BOLDE

BOLDE (Borjes Online Linguistic Development Environment) is a web application, a soft real-time collaborative program that is accessed with a web browser. Its function is to serve as a linguistic computational tool, where the user can develop computational linguistic resources (like formal grammars). This development process requires a few different capabilities, which BOLDE tries to provide, namely the creation, editing, storing, testing, and perusal of files.

BOLDE can be hosted remotely or locally. It is currently available online at the url (`https://garciasevilla.com/bolde`). However, it can be installed locally, and used as a local application. All the components of BOLDE are open source and are freely available on the Internet.

This chapter describes the implementation of the server and client components of BOLDE. It also explains how to install and run the application locally or in a server, and finally, the organization of the code and project is described.

4.1 Client-server communication

As explained in chapter 3, BOLDE as a web application has two main components, a server, hosted remotely (or optionally locally) and a client, the user's machine. Communication between the two components is realized with HTTP requests and responses, the basic mechanism of the web.

Static files are retrieved with the HTTP method `GET`. Browsers use this method to retrieve the entry point of the application at the BOLDE url, as is standard in the web. The client program is also served as a static file, consisting of the HTML presentation and the Javascript logic.

Logic requests to the server, which can be seen as "API requests", are performed with the `POST` method. For these requests, all the route urls are of the

form `'/api/<module>/<operation>'`. The request bodies use JSON (Crockford 2006) for encoding the transferred data. Since the pertinent operation and module are encoded in the url, the body of the request just needs to encode the arguments for the operation. Responses are also given in JSON encoding.

4.2 Server

The server software is written as a Javascript module for Node.js (*NodeJS* 2015). Node.js is a runtime for Javascript programs outside of the browser. It provides access to system resources not available in a browser environment, for example to the filesystem or to raw network input/output. It uses an event-driven model, where the program waits for requests to arrive, and then serves it in a non-blocking way. It is ideally suited for web servers and realtime web API's (application program interfaces).

4.2.1 Overview

The server is divided into a few different modules. Data flow, initiated by a user request, starts at the server entry point, `app.js`. If a static resource was asked for, it is immediately served. Document editing functions go to the `sharejs.js` module. If some server logic is required, the appropriate module is called. These modules may need to store or access persistent storage, which is handled by `store.js`. The data flow can be seen in Figure 4.1.

4.2.2 Libraries

The server leverages a number of existing libraries, the main ones listed below. The full list can be found in the file `package.json`. All of these libraries are distributed as `npm`[1] packages.

- **config**[2]. A library for merging default and user configurations.
- **express**[2]. A web server library.
- **fs-extra**, **fs-promise**[2]. Improvements on default filesystem functions.
- **js-yaml**[2]. A YAML (Ben-Kiki, Evans, and Ingerson 2009) parsing library.
- **log4js**[2]. A logging library.

[1] `https://www.npmjs.com`
[2] `https://www.npmjs.com/package/<package-name>`

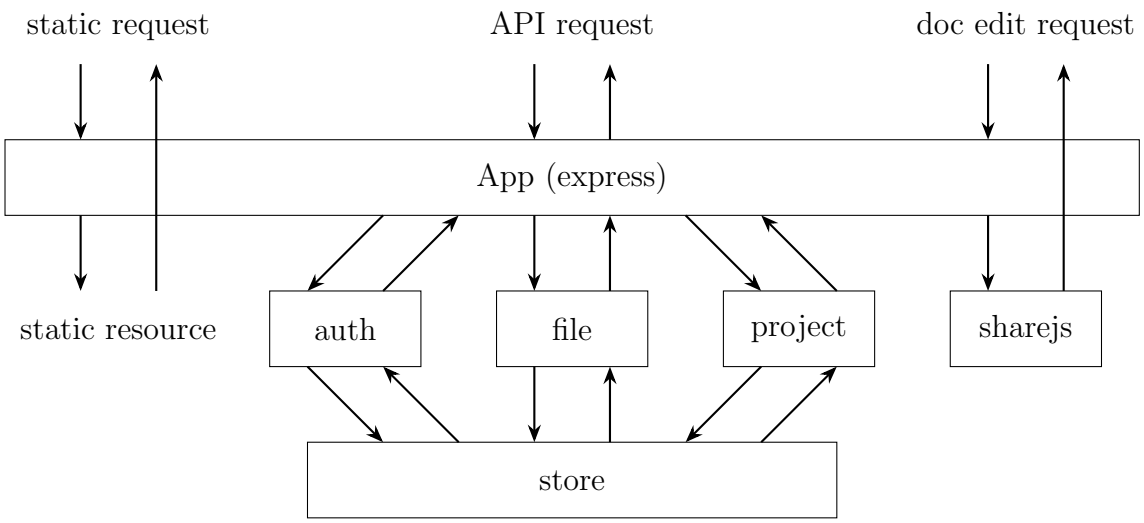


Figure 4.1: BOLDE server data flow

- **share**[2]. A concurrent document editing framework.

Among them, `express` and `sharejs` are the most important ones. They are in charge of substantial parts of the functionality, but their use model, and thus their use in BOLDE, are very different.

Express is what we might call an "embedded" library. The objects it exports are then specialized and made to work by the user code, which is responsible of properly putting the pieces together. The BOLDE module mainly interacting with express is **app**, the main entry point.

Share, on the other hand, is started by the user code once, and from that point onwards communication with it must be performed via its API. We could call it an "external" library. This communication is all performed by the **share** module.

Another library worth mention is `log4js`. It is used by all modules, when they need to record that a given operation was performed, or an unexpected situation happened. Log4js collects all these records (logs) and tidies them up, adding information like a timestamp, or the subsystem which produced them. It can be configured to output the logs to the console and/or to the filesystem.

This configuration, and other BOLDE configuration, is taken care of by the `config` library. See subsection 4.2.4 for more on this.

4.2.3 Modules

The communication between BOLDE modules is always in the form of Promises (Kambona, Boix, and De Meuter 2013). Promises are an eventful programming paradigm, where asynchronous operations are hidden within Promise objects. They store and handle callback functions, which are called when the assigned operations are completed. They can be chained, and easily passed around, making asynchronous programming easy and intuitive even in procedural paradigms.

When data enters the system as an http request, different modules may have to be called. These modules then return a promise, upon whose fulfilment or rejection, a response is sent to the client with the pertinent information. The promises encapsulate the global data flow from Figure 4.1.

4.2.3.1 app

The entry point of the BOLDE server is the "app" module, as per the usual convention in the node.js community. This module is in charge of preparing the rest of the modules, and loading the libraries. It finally starts an http server, with the help of the express library, and sets up all its routes.

In express, the logic in charge of handling an http request is called a Router. These routers can have different routes, which are urls or templates for urls, which when matched trigger a callback function. The callback function, in BOLDE, then calls the appropriate module, if any logic must be performed. If the request was for a static file, it is simply served directly. Node.js makes use of the Linux kernel capability to send files to the network directly from memory, without CPU intervention (sendfile, Alexander and Kuznetcov 2002). This makes static file serving very efficient and lightweight, improving response time of the application.

When the request for a logic operation is received, its module is invoked. It returns a promise, which represents the asynchronous completion of the requested operation. When it is fulfilled or rejected, the HTTP response is sent to the client, also encoded in JSON. The JSON response includes two fields, the first one being `ok`, a boolean value which signals whether the operation was completed successfully. If so, a `data` field is also included, with any resulting or requested data from the operation. If `ok` is false, an `error` field contains the error that caused the operation to fail.

4.2.3.2 Logic modules

The logic modules are in charge of the proper entities and functionality managed by the server. When the client interacts with the server via an API call (so not just requesting for a static resource), it is trying to perform an operation on a certain abstract entity. Each module manages each of the entity types, and sets up the appropriate routes in the main express server to handle the corresponding operations.

The first such module is **auth**, which is in charge of user authentication and settings. The entities it handles are thus the users, and the operation it exports is to log into the system.

A second module is **project**. The entities are the different user projects, and the operations it provides on them are creation, deletion, modification and, of course, retrieving.

Each user can have many projects, and each project comprises a number of files. These are handled by the module **file**, which exports the functionality for creating and deleting them, and for copying a file into another file. The editing of files is handled by a different module, **share** (see subsubsection 4.2.3.4).

This module architecture allows functionality to be added in an extensible way. Were a new entity to be added to the system, a new module would just have to be created and plugged into the server routes. And when new operations have to be defined on an existing entity, a new route is added, and the code written in a localized and modular fashion.

4.2.3.3 store

Even if having the logic divided into different modules is advantageous, there is some functionality which all share, and should be thus put together in its own module. This is the persistent layer, the **store**. The store module gives access to permanent storage to all the other modules. It exposes two different kinds of resources: user files and user metadata.

User files are the main editing entity in BOLDE. They are stored in the filesystem as actual files, and the store is in charge of retrieving and updating them. They are identified by a traditional path, separating directories with forward slashes.

User metadata, like project lists and user settings, are also stored in the system as physical files. They are identified by a resource name, and are stored in YAML format, which is readable and writeable by both computers and human beings.

The store module is in charge of loading and updating this metadata, hiding from the user the implementation details of the actual storage on the filesystem.

If, in the future, the filesystem should be substituted for another type of persistence layer (like a database), only the store module would have to be updated, since the rest of the modules depend on it and make use of its abstractions.

4.2.3.4 share

The final module in BOLDE, but not the least important, is **share**. This module is in charge of the (soft) real-time collaborative editing of documents. When it is initialized, it sets up the `sharejs` library, and routes a number of urls to it.

Of these urls, some point directly to the library, passing through the requests. These are the editing requests, such as "insert characters", "delete", "add element to list", etc. They are completely managed by `sharejs`, and it gives the response directly, bypassing the main server. These requests are expected to be generated by the parallel `sharejs` library in the client, and thus can be thought of as a separate communication channel.

The other urls are for requests generated from the BOLDE client. These urls are in charge of opening files, which belong to BOLDE, so that `sharejs` knows their contents; and also of saving the changes continuously. The `sharejs` library only takes care of synchronizing clients for collaboration, but it does not store information persistently. The way this is solved in BOLDE is by having the server act as another "client" to `sharejs`. This client does not generate any edits on the documents, but merely reads the contents. When these have been changed, they are saved to the store module. The BOLDE client periodically notifies the server about which files are open, so that it can close those that are no longer open in any edit session.

4.2.4 User files and configuration

We have seen that user files and metadata are saved by the store module to the filesystem. The location where they are stored is configurable, and by default is a directory `user_settings` in the same path as the BOLDE installation.

This path, and other configurable settings in BOLDE, are stored in the files under the config directory. The configuration file `default.yml` contains all the configurable parameters and their default settings. It is a human readable file, and all settings can be overridden by writing a `local.yml` file in the same directory, with the new values for the desired parameters. Omitted ones take the default

```
# The path for storing files
user_files: user_files
# Http server configuration
server:
    protocol: http
    hostname: localhost
    port: 3000
    path: ''
# Log configuration
log:
    appenders:
        - type: console
        - type: file
          filename: logs/bolde.log
          maxLogSize: 100000
          backups: 5
    levels:
        '[all]': 'TRACE'
```

Listing 4.1: Default BOLDE configuration

values. The default configuration can be seen in Listing 4.1.

Another file in the config directory is `welcome.md`. This is a `commonmark` (MacFarlane 2014) file, which is translated to HTML and served as the main landing page to every user of BOLDE.

4.3 Client

As was explained in the previous chapter, the BOLDE client is written in a set of different technologies, often referred to under the umbrella term of HTML5. In the particular case of BOLDE, however, most of it is written in Javascript. While web browsers require HTML for presentation, BOLDE only has one such file, the 19 lines long `index.html`.

This file, when loaded by browsers, loads itself the main application code: `bundle.js`. This file is a Javascript file which programmatically initializes the user interface and the client logic. The user interface is of course in the form of HTML, but is generated by a Javascript library: **react**.

Along with react, a number of additional Javascript libraries are again used by the BOLDE client, the most important listed below:

- **browserify**[3]. A library for packaging modular code into a single file for faster download.

[3] `https://www.npmjs.com/package/browserify`

- **flux**[4]. A library for implementing the flux design pattern.
- **less**[4]. A compiler of less files into css.
- **nedb**[4]. An in-memory document database, used by the treebanking module.
- **react**[4]. A library for building user interfaces.
- **tcomb**, **tcomb-form**[4]. A data-type library, useful for programmatically generating user input forms.

These libraries are distributed via the same channel as those of the server, `npm`. In fact, they are downloaded at installation time, along the server libraries. They are then compiled and repackaged into the `bundle.js` file, for serving to the client.

Another important library used in BOLDE is `Ace`[5], a code editor for the browser. It is used as the text editing component. It cannot be easily installed with `npm`, so it requires another tool: `bower`[6]. `Ace` is the only library that uses `bower`, which is another package distribution. Its use is being phased out in BOLDE, so that future versions will not need to depend on it.

4.3.1 Compiling and serving

While Javascript is an interpreted language, it is very low-level, and has very few abstractions to help with the development of a large project. At the server side, node.js as a platform provides some such utilities, like the loading of external modules with `require` function calls (*CommonJS module specification* 2015).

At the browser side, many solutions have appeared over the years. From higher level languages that are compiled to Javascript (Typescript[7], Coffeescript[8]), to improvements to the language, standardized by the ECMA (Ecma 1999). The tool used in BOLDE is an npm library called **browserify**.

Browserify takes Javascript code written in a modular way, using `require` function calls, just as in the server. It packages this code into a single file, which can then be served as a whole, for the browser to receive and interpret. While this has huge benefits in terms of network efficiency, there are also other advantages.

[4] `https://www.npmjs.com/package/<packagename>`
[5] `http://ace.c9.io/#nav=about`
[6] `http://bower.io/`
[7] `http://www.typescriptlang.org/`
[8] `http://coffeescript.org/`

As it traverses the dependency tree for the application, browserify can make use of other resources that are not Javascript. For example, it can take files written in a higher level language, and compile them on the fly to Javascript, before packaging them as if they were regular Javascript code. This is used in BOLDE for writing the user interface. This code is written as React JSX files, a higher level extension of Javascript, for easy generation of HTML. JSX also accepts many features of Javascript which are already standardized by ECMA but have not been implemented in many browsers yet.

Apart from the user interface logic, the visual style and appearance of the application are also compiled by browserify. They are written in the LESS language[9], which is an extension to CSS (Bos et al. 2005). The generated CSS code is then transformed by browserify, so that it can be imported from Javascript and given to the browser in the same bundle, for displaying the application.

4.3.2 Architecture

The main purpose of the client code is to provide an interface to the user, a point for interacting with the server. But since BOLDE uses a somewhat thin server architecture, the client has many responsibilities too. Within the development environment, the user is able to create, delete and edit files, organize them in projects, collaborate on these projects, and even execute pipelines.

As stated in section 3.2, this complexity is managed with the Flux design pattern, which basically eliminates two-way bindings, and makes data and action flow within the application explicit and unidirectional. The Flux pattern as implemented in BOLDE divides logic into three basic modules: Actions (which encapsulate the dispatcher), Stores and Components (the views). See Figure 3.2 again for their relations and the client data flow.

4.3.2.1 Actions

"Actions" objects are abstractions for events that initiate change in BOLDE. They are often started by the user, for example by clicking on a button or other user interface device. They can also be started by other parts of the application, for example by asynchronous running processes (the engines) or by the server itself, notifying of an update. Many actions require informing or consulting the server first, and indeed all communication with the server from the client has to be performed as an Action.

[9] `http://lesscss.org/`

However, actions are functions with no return value. They are simply methods that are called to start the appropriate data flow, and its results (for example, the server response) have to be retrieved in a different way. This is the global Flux dispatcher, which notifies all the parts in the application of the different movements of data. Stores listen to this dispatcher, and receive updates from it. When an action is emitted by the dispatcher, the appropriate stores modify their internal state to reflect the desired change that this action brings.

4.3.2.2 Stores

"Stores" keep the state of the application. Actions are only static methods for starting change, and Components render the user interface as a function of the state kept in the stores.

Thus, every piece of information about the user session must be kept in a store. There are stores which deal with server side objects, like files and projects, and which keep track of their contents and their relations. Other stores only deal with client state, like the Tab store. This store manages all the different views that the user has opened, where they are on the screen, and is in charge of opening and closing them, when the dispatcher notifies it that the corresponding action has been taken.

4.3.2.3 Components

Components are the final piece of the puzzle. They wait for stores to change, and when they do, they render themselves according to the new information. Components are the actual elements that are seen by the user in the browser window, and are the interface which recovers user input. When this input is found, an Action is started, and thus the data flow cycle is completed.

Since components have to be rendered on-screen, they need to be HTML objects. That is the language the browser understands, an HTML DOM (Document Object Model), based and not unlike an XML DOM. The purpose of the **React** library, which has been developed by Facebook[10] for their site and application, is precisely to allow developers to write modular components in Javascript, which are then translated to HTML for the browser to display.

React components are objects with some inner state (different from that of the stores, in that it is only about visual properties) and a few different methods. The most important method is `render`, which returns the HTML code to be sent

[10] `https://www.facebook.com/`

to the browser. It does not actually return raw HTML, however, but rather an intermediate representation, a lightweight DOM so to say, that React uses to update the browser in an intelligent way.

Instead of drawing the whole user screen every time there is a change, React collects the outputs from the components' rendering, and compares it to what is already on display. When it finds a change, it computes the transformation needed to realize it, and only notifies the browser of this change. This way heavy rendering of parts of the application do not stall the rest of the application from rendering or being active.

There are many components used in BOLDE. Some are small, covering only a discrete piece of user interactions, like the `Prompt` component. Others span the whole window, like the `TabPanel`. This component serves as the scaffolding for the rest of the components, which are embedded as tabs into the different panels. Most can be found somewhere between the two extremes, but all adhere to the Flux architecture, which improves maintainability and helps keep a consistent interface between the different implementation subsystems too.

4.3.3 Visual editor

An important Component in BOLDE is that of the Visual Editor. Most of the files in the development environment are edited as plain text, for which the text editing component, based on the Ace library is used.

But formal grammars can also be edited visually. This complies with the requirement that linguistic development can be done in the platform without the need to know how to code. It is also a user-friendly solution, which maps more intuitively to the way an expert might think about the grammars being developed.

The visual editor offers an interface for creating and updating the different elements that can be found in a formal grammar. The class of formal grammars it allows to edit are those that can be used by the linguistic library that BOLDE uses, which will be discussed in the next chapter.

However, that library is flexible and unassuming, and offering all of its power would require a complex and unintuitive interface. It was thus decided that the visual interface would be a bit more opinionated. While it still offers most of the underlying functionality, it displays and organizes it in a way which is typical of HPSG grammars. Nonetheless, any kind of grammar which is based on unification, and which uses context-free rules, can be encoded in it.

The visual editor displays the objects in the grammar as graphical objects,

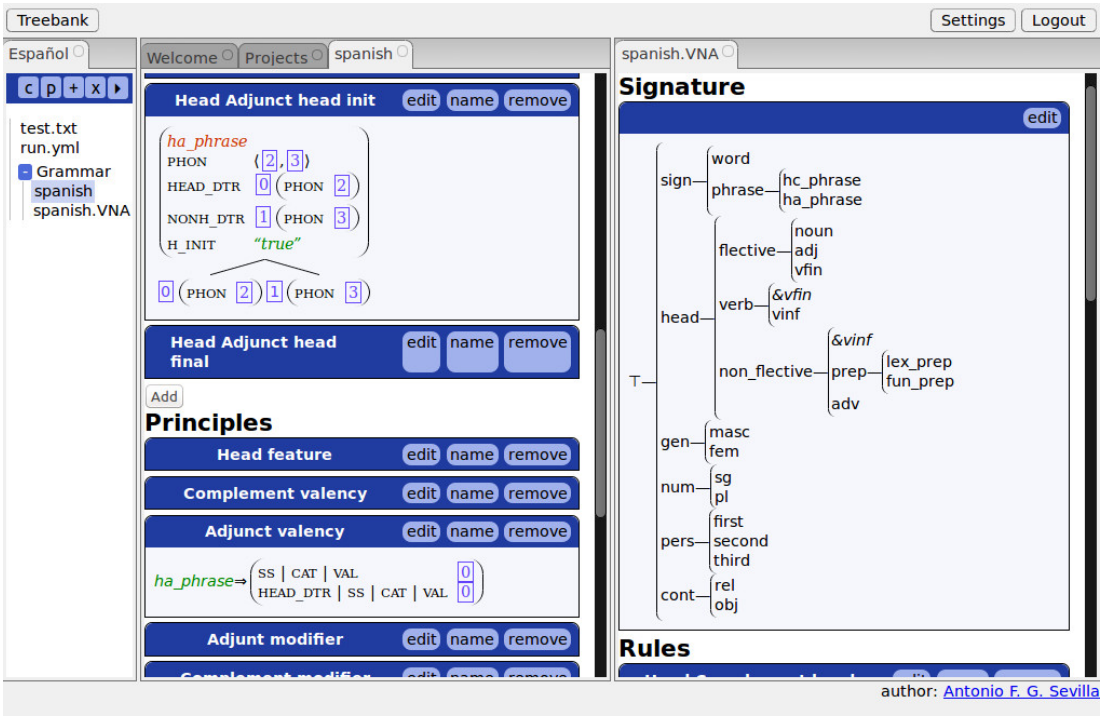


Figure 4.2: Screenshot of the visual editor

to which it attaches buttons and editable components. This way the user can visually manipulate and create grammatical information, like phrase structure rules, constraints, or lexical entries.

The lexicon editor is a very useful addition to the visual editor. Called Paradigm Editor in code, it is a component which allows to add entries to the grammar's lexicon in a visual, powerful and intuitive way. Morphological information can be easily input, and entries for words of the same class can share the same visual object, so that information is clearly structured and not repeated unnecessarily.

The Paradigm Editor allows the user to add a number of grammatical templates to a paradigm. These are representations of the surface forms of entries in the lexicon, and can be edited visually. For each entry, a morphological rule can be added. This rule can inflect the lemma, via regular expressions, to create different word forms. Regular expressions allow simple agglutinative morphology to be encoded, but also arbitrarily complex fusional morphology. They are almost equivalent to finite state transducers. What they miss, such as some parametrization, is provided by the Paradigm Editor in the form of template guards, called "classes" in the interface. Each lexical entry can have a class, which might be a structured linguistic object (a feature structure) or a simple literal. Only if this class unifies with the class for the template is this morphological rule used.

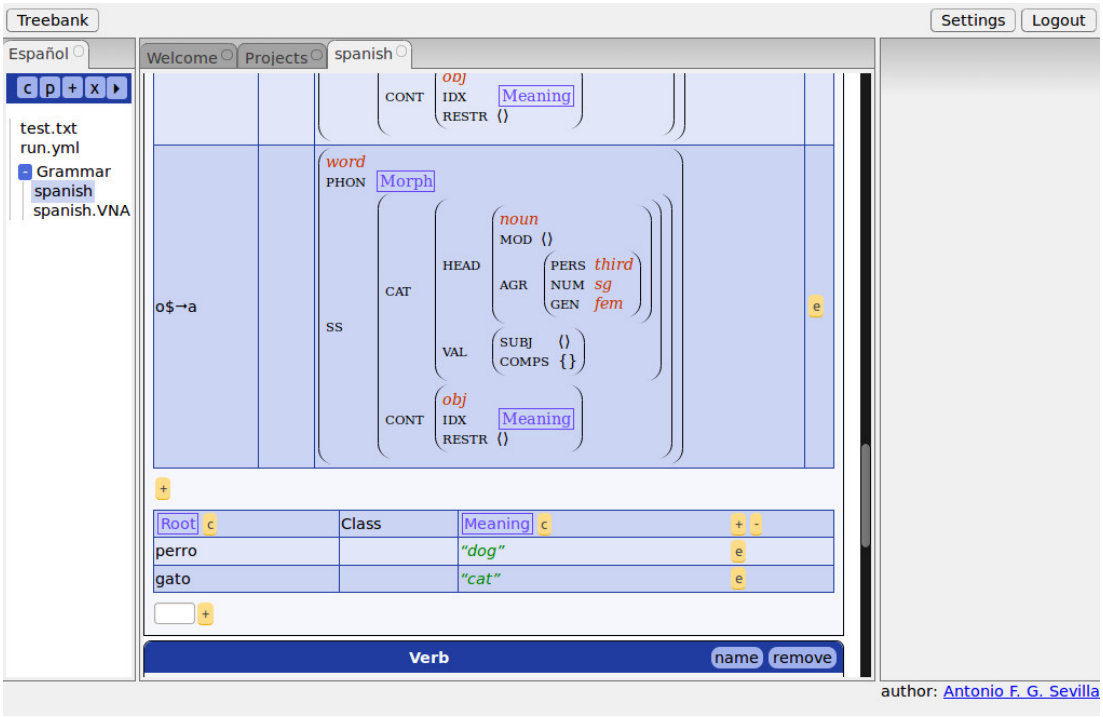


Figure 4.3: Screenshot of the paradigm editor

This can help with "regular irregulars", or just to further subdivide an inflection paradigm.

Apart from the class, and a "root" or dictionary form, each entry in the paradigm can have any number of additional parameters. These parameters can then be used in the template for encoding different information. For example, these might be case, gender or number information. In the paradigm's template, these parameters will be encoded in the appropriate formal way. However, in the listing of lexemes, each one will be displayed as a table row, with only the relevant parameters to its side. This resembles a traditional linguistic inflection table, and is a very natural, fast, and efficient way to encode lexicon information.

4.4 Engines and Pipelines

The final part needed for the system is the execution manager, which will actually make the files in the project useful and usable. As stated in chapter 3, the user should be able to configure different pipelines of execution, where data is input into the system, and different engines process it and pass it around until the final result is obtained.

The pipeline for each project is written by the user into a `run.yml` file, in YAML format. The different components can be instantiated, and their outputs

and inputs connected, so that with a simple invocation of "run" (by clicking the start button) all processes are automatically run. An example configuration file can be seen in Listing 4.2.

The different pipeline components can be:

- **filesource**. This element takes the contents of a file and puts them into the data flow.
- **filesink**. The opposite to the filesource, the sink takes all the data it receives and stores it in a file.
- **tee**. Splits or merges the data flow. It is the only element that can be listed more than once in the connect section. All the input that it gets, from any connected sources, is output replicated to all connected sinks.
- **javascript**. This element receives a Javascript file as configuration. This Javascript can contain arbitrary code, which is run for each line of input received. The input data is in the variable `input`, and output is performed with the function `output`. State can be kept between function calls, by writing properties to the `state` object.
- **borjes**. This element runs the linguistic engine, configuring it as a parser. It must be given as configuration the type of grammar to use (CFG/HPSG), and the grammar file to interpret. It then parses all lines in the input, and outputs the parse trees for them.
- **display**. Visually displays the output from the parser, in the form of a list of trees.
- **treebank**. Stores the result from the parser into a tree database that can be later searched.

Of these elements, three are proper "engines": javascript, borjes and treebank. Their operations can be quite lengthy, and take some time to complete. In order for the application not to hang during engine execution, these engines are forked to the background by BOLDE. This is doable thanks to a relatively modern feature of HTML5: Web Workers (Järvinen 2011). Web workers allow a script to be executed in a separate context from DOM rendering (and thus, user interface); effectively simulating what in traditional systems programming would be called a forked process.

```yaml
# Elements can be first listed, for easy reference afterwards
elements:
- &input
  type: filesource
  files:
  - test.txt
  - bad.txt
# The borjes component is the linguistic engine
- &lcfg
  type: borjes
  config:
    format: CFG
  files:
    grammar: english.yml
- &filter
  type: javascript
  files:
  - filter.js
- &display
  type: display
# Tees allow the pipeline to be split, so that data can be merged
# into a single channel, or sent simultaneously to more than one
# different components
- &tee
  type: tee
- &treebank
  type: treebank
  files:
  - result
# The connect section can list many different linear pipelines.
# Here, the input is parsed by the linguistic engine and then
# displayed, but before that, it is also sent to a treebank.
connect:
- [ *input, *filter, *lcfg, *tee, *display ]
- [ *tee, *treebank ]
```

Listing 4.2: Example of a run.yml file

4.5 Installation

BOLDE can be distributed as source code, and it also lives in a private git repository, to which access can be provided if requested. Once it has been downloaded to the filesystem, it is very simple to configure and start the application.

First, `node.js` must be installed. A recent enough version (≥ 0.12) is required, and might have to be compiled from source. GNU Make is also required, to orchestrate the different actions needed for setting up the application. The first one, `make update` will download the rest of the dependencies. Then, a plain invocation of `make` will compile the source, and make it ready for running.

The configuration file `config/local.yml` should be created, and any settings

from `config/default.yml` that need to be overridden should be written there. Finally, a `config/welcome.md` file should be created, and a welcome message provided.

To start the application, npm (installed automatically with node) is used. Being in BOLDE's root directory, where `package.json` is, it is enough to call `npm start`. This will start the server, and the BOLDE client will then be available at the location specified in the config file. To create a user, a request must be directed to the API which is not available from the client. With curl, it must be: `curl <bolde url>/api/newuser -d '{ "user": "<username>", "password": "<password>" }' -H "Content-type: application/json" -X POST`.

4.6 Structure of the code

BOLDE code is organized in a number of directories, listed below.

- **root (.)**. General configuration files, and the Makefile are kept in the BOLDE root. This is where the user files directory has to be located (if it is a relative path), and from where the server should be started.
- **static**. Static resources, such as the index.html page and images, are kept here.
- **config**. The application configuration files are kept here.
- **server**. Server code is all kept under the server directory.
- **src**. The code for the BOLDE client, which must be compiled, is stored in the src directory
- **build**. Build results for the client end up in this directory.

4.7 Summary

BOLDE is the main software solution to our problems of chapter 1, in the form of a web application. It has the user interface, and the editing and collaborative capabilities that were required. But a crucial component is missing. BOLDE has to be a linguistically-savvy platform, a tool for doing computational linguistics. To meet this objective, the linguistic library Borjes has been developed, and is described in next chapter.

Chapter 5

Borjes

Borjes is a programming library for parsing text according to a formal grammar. It is written in Javascript, and can run in both client (web browser) and server environments. It is written in the standard ECMAScript (ECMAScript, Association, et al. 2011). As a library, rather than providing a tool that reads grammars or parses text, it supplies developers with objects and functions that perform these tasks.

Borjes can be used for any parsing of text which requires a formal description. It provides objects for building this description, and functions for using it in parsing. The underlying paradigm is unification, but it is a paradigm powerful and expressive enough that it can be used to build any type of formal grammar, so long as it makes use of context-free rules and constraints. While the nomenclature is largely inspired by HPSG, translation to other grammar formalisms is straightforward, and the objects provided by Borjes make no assumption as to how the formalism operates. In that respect, Borjes can be said to be a generic formal grammar library.

5.1 Overview

Borjes is structured as an object-oriented library. It is organized around a well defined type system, and has a number of common patterns for making use of the different modules. The main objects are referenced in Figure 5.1.

All objects and modules in Borjes are exported as properties of the Borjes module. Top level objects, such as the Parser and Grammar, can be accessed directly. Access to the unifiable type system is performed via the `types` property, which exposes all constructor methods and the utility functions.

To create Borjes objects, constructor functions are provided. These are not

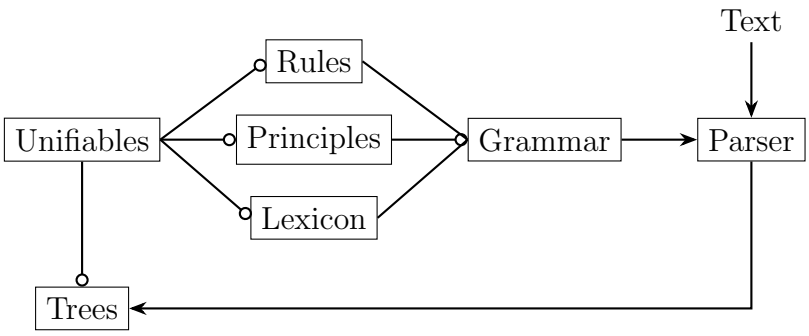


Figure 5.1: Borjes architecture overview

called with the `new` keyword, so are not true objects, but rather are given the parameters with which to create the designated type. Class methods are all static, and are called as such, passing the object on which to act as a parameter (see Listing 5.1).

This chapter is dedicated to explaining the library, its use and implementation. It first covers a fundamental design decision, that of the library's underlying data format, the "POJO". Then the library type system and innards are explicated, including code examples of their use. The last few sections explain how to use the library as a developer, and how the code is structured. A final section is included on a complementary library, Borjes-react, which allows graphical editing and display of Borjes object in HTML.

5.2 Why POJOs

As mentioned, Borjes objects are not class instances, but Javascript "objects" without a prototype chain. This is what is commonly called a "Plain Old Javascript Object", or POJO. It is one of the native types of the language, and represents a dictionary data structure, a mapping from string keys (called prop-

```
1 var Borjes = require('borjes');
2
3 var World = Borjes.types.World;
4 var myworld = World();
5 World.put(myworld, somevalue);
6
7 var Rule = Borjes.Rule;
8 var myrule = Rule(mother, daughters);
9 var result = Rule.apply(myrule, some_objects);
```

Listing 5.1: Example of the use of Borjes modules

erties) to arbitrary values.

While Javascript gives the possibility of using prototypal inheritance, useful for a more full-featured simulation of object oriented programming, the language itself is not object oriented. It was decided that instead of using these features, Borjes would only use Plain Old Javascript Objects for a number of reasons.

On the one hand, this simplified the handling of nested objects, and made possible the (simulated) use of both pass-by-reference and pass-by-value to functions. These features of a programming language, not native in Javascript, are crucial for designing a more rigid type system than what it offers.

On the other hand, plain objects have a very advantageous property: they are one-to-one serializable. This means that they can be described in plain text (Crockford 2006) without the loss of any implementation detail, making the inputs and outputs of every computation both extremely portable and human-readable.

From an implementation point of view, one can think of plain objects as data structures, with a number of fields that can take different values. While Javascript allows the fields to change dynamically during execution (they are proper dynamic dictionaries), modern execution environments take advantage of code that uses them as fixed data structures, and heavily optimizes the access to the keys (Ahn et al. 2014). Thus, a type system defined by data structures can be implemented on top of plain objects, and that is precisely how Borjes operates.

5.3 The type system

As mentioned, Borjes objects are plain dictionaries. Their main feature is the type tag, a string under the key `borjes`, which defines what kind of object they are. This tag defines what other properties will be defined for the actual Javascript object, and also tells the system how to work with it.

There are two main classes of Borjes objects, unifiable and non-unifiable. Unifiable objects represent pieces of grammatical data, like a word, its case, or a phrase. Non-unifiable objects represent grammatical functions, the rules that make up the grammar, or objects with related functionality (like the parser).

5.3.1 Unification

Unification is a term that stems from logic and computer science, and refers to a process of equation solving. It works by restricting the possible values of free variables so that a number of equations may be simultaneously true.

Due to its origin in logical frameworks, unification is generally understood in these abstract terms, of variables and equations. It is used as such in logical frameworks like ALE[1], based on Prolog, where it is the main process by which a program (stated as a system of equations) converges to a solution (its result). Formal grammars, especially HPSG, are usually implemented in these terms.

In spite of this, in imperative programming languages no such process exists natively. It could be implemented on top of them, effectively creating the basis for a Prolog interpreter or compiler. It is more useful, however, to use the native constructs of the language instead. Indeed we see that some of them actually correspond to the higher-level constructs used by the grammar itself, so some advantage is gained.

For example, feature structures have to be somehow defined in the logical paradigm of variables and equations. They are usually interpreted as functions from keys to values, and so a unification paradigm that supports unification at a sufficiently high level must be used.

In an imperative programming language, feature structures are just common data structures, and this native data type can be easily leveraged. In Javascript in particular, POJOs (see before) are particularly suited to creating this kind of construct.

What this means, however, is that unification must be defined in this new paradigm. It makes no sense to try to stick to the formal definition, when there are no true free variables or any equations, and thus in Borjes (and in this document) the term "unification" is used to describe a process which roughly mimics the logical unification, and which produces equivalent results in terms of grammar writing.

Borjes unification takes two objects and tries to find another, which has a value that is compatible with both of them. What this compatibility means depends on the type of the objects, and is why the type system is so central to the library. It could be argued that "matching" would be a more appropriate name for this process, and indeed it is called that way in programming languages like Haskell. However, since it is used in the same way and for the same reasons as in the implementations in logical frameworks, the term "unification" shall remain.

5.3.2 Unifiable types

Objects of a unifiable type represent pieces of grammatical information. They are defined in terms of two characteristics: what kind of information they store,

[1] `http://www.cs.toronto.edu/~gpenn/ale.html`

```
var types = require('borjes').types;
var unify = require('borjes').unify;

var Nothing = types.Nothing;
var Anything = types.Anything;
var Literal = types.Literal;

unify(Nothing, Nothing); // -> Nothing
unify(Nothing, Anything); // -> Nothing
unify(Anything, Anything); // -> Anything
unify(Literal('John'), Literal('Mary')); // -> Nothing
unify(Literal('John'), Anything); // -> Literal('John')
unify(Literal('John'), Literal('John')); // -> Literal('John')
```

Listing 5.2: Example of the use of primitive types

and how they unify. Listing 5.2 shows how to use these types in Borjes.

5.3.2.1 Primitive types

Primitive types represent a single, constant piece of information. They can be copied by reference, since all objects of the same type and value are interchangeable. They should not be modified dynamically.

- **Nothing**. The nothing object represents non-existence, absence of possible values. It is used to indicate failure of unification. It does not unify but with itself. Sometimes called "bottom" in logical formalisms.
- **Anything**. The anything object represents lack of information about possible values, meaning any other object is compatible with it. It thus unifies with all other types, and the result of unifying any x with Anything is precisely x. Often called "top".
- **Literal**. A literal represents a single, atomic concept, implemented as a string. It can only unify with other literals of the same value. It can be used for representing words.

5.3.2.2 Lattices

Lattices are a kind of global objects, which are shared by all the code in the same execution context (what would be called static in compiled programming languages). They are created with a name, and can be used anywhere with that name. While lattices do not unify, since they do not represent any actual

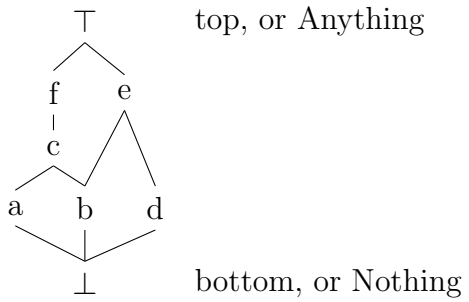


Figure 5.2: The lattice created in Listing 5.3

concrete information, they are presented here because their purpose is to describe how other objects unify: the elements of the lattice.

Lattice elements represent, like literals, constant and atomic concepts. They are also represented internally as strings, and that is how they are identified by the user. They are also primitive types. However, they unify in a richer way than literals, as described by the lattice to which they belong.

The lattice describes a hierarchy of its elements, a partial ordering in their universe. This ordering can be thought of, from the point of view of unification, in terms of specificity. If an element x is more specific than y, then the value of x is compatible with that of y (but not the other way round). Thus, unifying x and y would give y as result.

The usefulness of the lattice is that some elements might be more specific than more than one element, and there might be more than one more specific elements; and these specifity links might cross and overlap, creating a rich structure of hierarchical objects. For example, if z is more specific than x and y both, unifying x and y would yield z as value. If there are many possible unifiers, per definition the correct one is the most general (least specific). If the lattice is not bounded complete (there is more than one most general unifier), the result is undefined, so only bounded complete lattices should be used. An phony element might be used, representing the disjunction of the possible most general unifiers.

The common use for lattices in a grammar is to represent a type hierarchy. Not to be confused with Borjes internal types (which we are now describing), a grammarian might want to describe things such as nouns and verbs. They might at first use literals for this, but there are also proper nouns, countable nouns, transitive verbs, intransitive verbs... The grammarian might want to describe phenomena which apply to all verbs, but some that apply only to transitive ones. By using lattices, these relations, and more complex ones, can be encoded. The

```
var Lattice = require('borjes').types.Lattice;
var unify = require('borjes').unify;

// Creates a lattice from a hierarchy of names
// The lattice is the same as in figure 5.2
var l = Lattice.fromProto({
    f: {
        c: {
            a: null,
            b: null
        }
    },
    e: {
        b: null,
        d: null
    }
});

// Retrieve the elements by their name
var a = Lattice.element('a');
var b = Lattice.element('b');
var c = Lattice.element('c');
var d = Lattice.element('d');
var e = Lattice.element('e');
var f = Lattice.element('f');

unify(c, e); // -> b
unify(f, e); // -> b
unify(a, e); // -> Nothing
unify(f, c); // -> c
unify(e, d); // -> d
unify(f, d); // -> Nothing
unify(a, d); // -> Nothing
```

Listing 5.3: Example of the use of lattices

code in Listing 5.3 shows this in an example.

Another view of lattices, common in mathematics, is as representing a family of sets, with the partial ordering meaning inclusion rather than concreteness. Borjes takes advantage of this alternative point of view, and uses it for the internal implementation. Lattice elements are represented as bitmasks, each bit corresponding to an element. Apart from its own representing bit, the bitmask for every element has all the included sub-elements bits set on too. Thus, finding the most general unifier of two types is as simple and efficient as performing a bitwise `AND` operation.

5.3.2.3 Composite types

Objects of composite types are made of other Borjes objects, constituting recursive data structures. They allow grammatical information to be structured, and some have special unification semantics which are helpful to the grammar writer.

The most basic composite type is the feature structure, which holds a number of sub-objects for a corresponding number of features, or names. It is the most general and flexible type. The other Borjes types could be built by using feature structures, and indeed formalisms such as ALE work like this. In Borjes, it was decided to take advantage of the data structure possibilities of imperative languages to simplify grammar development.

Making the underlying system understand data types, instead of forcing users themselves to define them, has a number of advantages. Apart from the speed and optimization possible by writing unification rules directly in the implementation language, the system can understand what the user means or wants and help with development. This is most useful in visual aids and graphical interfaces (see section 5.5). Since the system knows about some of the different data types, it can display them visually in a more straightforward way that hides the implementation details.

Nonetheless, if the user were to need an extension to the Borjes type system, it would not be too complicated to implement making use of lattices (for class definitions), feature structures (for information structuring) and principles (for enforcing of constraints and properties).

5.3.2.4 Feature structures

Feature structures store different pieces of information in an organized way. Each piece of information is itself a Borjes object, and is kept under the label of a feature. They can be viewed as multidimensional vectors, where the different coordinates are indexed by a string understandable to humans, the feature name. Another point of view treats them as functions from feature names to values. These are all equivalent, and the important characteristic of feature structures is how they unify.

The unifier of two different feature structures is, again, a feature structure. Its value must be compatible with each of the original structures, the most general value that is at least as specific as both of them. It can be easily defined in a recursive way: the values for the features of the unifier structure are the unifiers of the values of the original structures. It does not matter what those values are, the only requirement is that they unify. If all of them unify, then we can say

```
var FStruct = require('borjes').types.FStruct;
var Literal = require('borjes').types.Literal;
var unify = require('borjes').unify;

var john = Literal('John');
var nom = Literal('nominative');
var acc = Literal('accusative');

var johnStruct = FStruct({ string: john, case: nom });
var nominativeNoun = FStruct({ case: nom });
var accusativeNoun = FStruct({ case: acc });
var recursive = FStruct({ modifier: FStruct({ string: Literal('
    long') }));

unify(johnStruct, nominativeNoun); // -> johnStruct (or
    equivalent)
unify(johnStruct, accusativeNoun); // -> Nothing
unify(johnStruct, recursive); // -> { string: john, case: nom,
    modifier: { string: 'long' }}
```

Listing 5.4: Example of the use of feature structures

that there is a feature structure, compatible with and more (or equally) specific feature by feature than the original feature structures.

The power of feature structures is that they can be underspecified in another way. Apart from having values which are not maximally specific, there can be features whose values are not specified at all. This is equivalent to having a value of `Anything` for those features, but without the need to list them explicitly. On top of convenience, this allows extensibility and modularity in the grammar. A rule can, for example, act on anything which has a valence, and not specify any of the other features. This way, different types of grammatical objects can be affected by this rule, while being completely different in the structure of their features. All of this can be seen in Listing 5.4.

5.3.2.5 Typed Feature structures

An extension to feature structures comes with typed feature structures. These are a very useful implementation-level object in unification-based grammars. They are feature structures with an additional characteristic: a type. In HPSG and other logical formalisms, feature structures are organized as a type hierarchy called "the signature". On top of it, a well-typedness principle is often either explicitly or implicitly applied, which ensures that feature structures of a concrete type have the correct features and their values are of the correct type. This has not always been the case, however, and it has even caused some complications

```
// continued from the lattice example

var TFS = require('borjes').types.TFS;

var eTFS = TFS(e, { name: 'John', surname: 'Doe' });
var fTFS = TFS(f, { surname: 'Doe' });
var aTFS = TFS(a); // a type restriction

unify(eTFS, fTFS); // -> TFS(b, {name: 'John', surname: 'Doe'})
unify(aTFS, fTFS); // -> TFS(a, {surname: 'Doe'})
unify(eTFS, aTFS); // -> Nothing
```

Listing 5.5: Example of the use of typed feature structures

(I. A. Sag 2003).

In Borjes, well-typedness is not enforced, and a single signature is not required. The type of typed feature structures can be any arbitrary Borjes value, for example literals. It is of course most useful, however, to have types belonging to a lattice-like hierarchy, which is equivalent to traditional HPSG signatures.

Apart from the special `type` value, typed feature structures behave in all other respects as regular feature structures do. Their use, as shown in Listing 5.5, is very similar to them too. One could then ask what is the advantage of having a different Borjes type, against just adorning all feature structures with an additional feature called 'type'. While it is equivalent in most regards, there are two main advantages.

First, the implementation always tries to unify the types of typed feature structures first. Thus, it can speed up unification by immediately discarding objects with the wrong type. With regular feature structures, the ordering is irrelevant, and expensive recursive unification could be performed before realizing that the objects are not type-compatible.

Another advantage is that, by telling the system that `type` is a special value, more salient than the other features, the system can use this knowledge to aid the user in the development process, for example by displaying it in a special way. A common case is that of type restrictions, which are often used in HPSG. With type restrictions, the only constraint that the grammar applies on an object is that it is of the correct (or compatible) type. This can then be displayed by the graphical interface in a more concise way, without drawing all the visual apparatus of regular feature structures.

```
1  var List = require('borjes').types.List;
2  var unify = require('borjes').unify;
3
4  var empty = List();
5  var john = List(Literal('John')); // one element
6  var johnSomething = List(Literal('John'), Anything); // at
       least one element
7  var somethingLoves = List(Anything, List(Literal('loves'))); //
       notice the recursive use of the list constructor
8
9  unify(john, empty); // -> Nothing
10 unify(john, johnSomething); // -> Nothing
11 unify(john, somethingLoves); // -> Nothing
12 unify(johnSomething, somethingLoves); // -> <'John', 'loves'>
```

Listing 5.6: Example of the use of lists

5.3.2.6 Lists

Lists are traditional logical programming data structures. They are somewhat flexible, but above all they are simple, and they are ideal for the use with recursive algorithms. In Trale, lists can be defined using feature structures and the signature. In Borjes, they are provided as native objects.

Lists store two pieces of information, a head, called `first` in Borjes; and a tail, called `rest`. In LISP these would be `car` and `cdr`. Usually, the tail is expected to be another list, thus allowing the list to grow indefinitely in a recursive way. The mark of the end of the list is the empty list, often a separate type or value.

In Borjes, the `rest` can be any Borjes value. This allows binding variables (see subsubsection 5.3.2.7) to the rest of a list, like one would do in Prolog, in order to decompose it or merge it. Other values, apart from variables or lists, for the rest feature, would not be so useful, reducing the list to a mere 2-tuple (pair) structure.

In Borjes, the empty list is a special, primitive, value, and it is used both to mark the end of a list and to distinguish between a list with no content and a list with *any* content (marked with an Anything object).

Of course, lists unify recursively, and the unifier of two lists is another list whose `first` element unifies the `first` elements of the lists, and the same with the `rest` value.

5.3.2.7 Worlds and Variables

Until now, we have merely listed some very simple data structures, from the point of view of an imperative paradigm. But the real power of unification and logical

paradigms come with the use of variables.

Variables in these formalisms are more like their mathematical counterpart, rather than mere stores for information as they are in imperative paradigms. Variables can be free, meaning they do not yet have a value, or they can be bound. And when they are bound, their value is the same everywhere. If some part of the grammar says that the value bound to a variable is more specific than was previously thought, then this is true everywhere else in the grammar. It is not a local change in value, but rather a global increase in information.

Of course, in a computer this abstract behaviour has to be simulated. Logical programming languages and systems provide this natively, since it is central to their framework. Imperative systems do not have this feature, since it does not really fit with the sequential operation paradigm. Borjes must thus implement it itself in order to allow the expressive power of equations and variables to be harnessed.

The first thing needed is a place to store the information pertaining to variables. In Borjes, this is realized in the `World` object. Worlds are non-unifiable types, similar to lattices, which determine the behaviour of the concrete unifiable type: the `Variable`.

Unlike lattices, however, worlds are not global objects. There can be an arbitrary number of worlds, and they can be copied around, created, shared and destroyed. All variables belong to a single world, and are actually mere pointers to an object living inside of it. Worlds store these objects, which are the values to which some variable is bound.

Apart from variables belonging to a world, all Borjes objects can be bound to one. This means that they, and all their sub-elements, live in this world. While regular objects are not affected by this, composite objects need to know in what world they live, so that they can have variables as sub-objects. All variables belonging to the same top-level object have to live in the same world, the one bound to the top-level.

When two objects which live in different worlds are unified, a new world must be created. This is the world which unifies the worlds of the two objects, and where variables in the unifier will live. If there were variables in the original objects, there will be variables in the unified object. These new variables will have as their value the unification of the values of the original variables, with their counterpart in the other object, which might or might not be another variable.

While this is all very abstract and complicated-sounding (and actually, quite complicated to implement) variables are best explained by their use, which is

```
var World = require('borjes').types.World;
var Variable = require('borjes').types.Variable;
var unify = require('borjes').unify;

var w1 = World();
var w2 = World();
var bfs1 = FStruct({ cat: 'noun' });
var bfs2 = FStruct({ phon: 'john' });
World.bind(w1, bfs1);
World.bind(w2, bfs2);
FStruct.set(bfs1, 'phon', Variable(w1, 'john'));
FStruct.set(bfs2, 'cat', Variable(w2, 'noun'));

var unified = unify(bfs1, bfs2);

var cat = FStruct.get(unified, 'cat'); // -> Variable
var phon = FStruct.get(unified, 'phon'); // -> Variable

// unified will be bound to a new world, 'newworld'
var newworld = unified.borjes_bound;

World.resolve(newworld, cat); // -> 'noun'
World.resolve(newworld, phon); // -> 'john'
```

Listing 5.7: Example of the use of worlds and variables

quite straightforward and intuitive.

Variables hold a single piece of information, their bound value. What makes them special is that they can be copied, and there might be more than one clone of them living under the same top-level object. This allows information sharing and reentrancy in Borjes objects.

For example, let us look at a typical grammatical example, that of agreement. In languages with gender as a grammatical feature, constituents of a nominal phrase must agree in gender, meaning they all have the same value for it. This could be implemented by saying that a nominal phrase is only found by the parser when all objects have the same gender. A different rule for nominal phrases should then be created for each gender, one listing masculine, other feminine, and so forth. With variables, a single rule can be formulated, where the value of the gender feature of all relevant words in the phrase is the same variable. That way, if the head of the phrase (usually a noun) has a masculine gender, then the value of the variable is bound to masculine, and everywhere it appears this value must hold.

This looks convenient, but the power of variables is beyond simplifying rule writing. The binding between the gender values of the nominal phrase will remain thanks to the value. If, upon further inspection (in the form of unification), the

gender of some adjective is found to be 'masculine animate', then automatically all other objects have this gender too. And if one of them is incompatible with it due to some other grammatical effect, then unification will fail.

Basically, if we look at composite Borjes objects from the point of view of graphs, with each node representing a feature value, variables allow cycles to happen in the graph. If we look at them as classical data structures, variables are akin to pointers, allowing more than one feature to share the actual same physical value.

5.3.2.8 Disjuncts

Now that we can use variables, we would like to put them together into equations. While arithmetic equations usually involve an expression being equal to another one, in logical formalisms the common way to state them is as predicates which must be true. For putting together our equations, we have the different data types, which structure information, and variables, which make it shareable. Unification then acts as a sort of conjunctive operand, making sure some facts are true simultaneously, maybe refining the facts so that they are compatible.

The logical counterpart to conjunction is disjunction, whereby a number of different alternatives are provided, and only one of them needs to be true (but more than one can). However, disjunction is very complicated to implement. In some frameworks, like LKB, it is actually disallowed (Oepen 2002). An alternative is to use the signature, with its expressive power, to represent thing that might take a number of different forms. When unifying, the elimination of invalid disjunctives is simulated by finding more specific values in the lattice, which only represent a subset of the original alternatives.

This does not provide the full power of logical disjunction, of course. In TRALE, the full logical power of Prolog is available, which includes disjunctives. Disjunctions allow the grammar writer to represent a fact which is typical of natural languages: ambiguity.

Borjes provides a native data type, the `Disjunct`, which is composite and unifiable. A disjunct has a list of alternatives, and upon unification may take any of them. Its use is fairly straightforward, see Listing 5.8 for an example.

Simple as it may seem, the reason why some implementations do not allow for disjunction is that it makes things very complicated, implementation-wise. It is also not very efficient, and must be used sparingly. All this is because when a disjunction is unified, the current world is split. Variables no longer hold only one value, but a set of different possibilities which depend on which disjunctive path

```
// continuing the lattice example

var Disjunct = require('borjes').types.Disjunct;

var e_or_f = Disjunct(e, f);
var a_or_b = Disjunct(a, b);

unify(e_or_f, d); // -> d
unify(a_or_b, e_or_f); // -> a or b

var world = World();
var e_or_var = Disjunct(e, Variable(world, f));
World.bind(world, e_or_var);
unify(e_or_var, a); // -> a variable with value a
```

Listing 5.8: Example of the use of disjuncts

is currently true. And if a variable is embedded within a disjunction, then it has only one value, but this value depends on the path taken. Besides, disjunctions can be embedded within disjunctions.

In a logical implementation, implementing disjunction might require backtracking, an expensive and complicated procedure. In interpreted languages with reflection capabilities, that might be the best bet, storing stack frames and relaunching execution threads in them.

However, most languages do not offer this capabilities. Javascript certainly does not, and another solution must be found. The way it is implemented in Borjes uses a mechanism similar to backtracking, but more primitive and suitable for imperative paradigms.

The Borjes unification module keeps a context object during unification, for book-keeping and mapping variables correctly from old and new worlds. It also keeps a stack, and uses it for unifying values repeatedly, taking different disjunctive paths every time. That way, at the end of unification, every possible path has been unified or rejected, and a list of alternative unifiers is given to the user. In grammars with no use of disjunction, this stack should give no noticeable overhead to the speed of unification, but allows those who do use disjunctives to use them anywhere in the grammar where a Borjes object is valid.

5.3.2.9 Sets

Another type of objects which make use of the unification stack are sets. Sets are collections of Borjes values, in which order does not matter. They are slightly different from mathematical sets in that membership to a set is not necessarily unique, therefore allowing an object to be in a set more than once. The more

```
1  // continuing the lattice example
2
3  var Set = require('borjes').types.Set;
4
5  var set = Set(a, c, d);
6
7  var world = World();
8  var dsum = Set.sum(Variable(world, e), Variable(world));
9  World.bind(world, dsum);
10 unify(set, dsum); // -> there will be two possibilities:
11     // 1) the first variable has value b, the second Set(a, d)
12     // 2) the first variable has value d, the second Set(a, c)
```

Listing 5.9: Example of the use of sets

correct term for them might be "bags".

A way of implementing sets, in a framework which does not offer them natively, would be to use lists. In lists, order matters, but in sets it does not, so it is irrelevant whether the underlying data structure keeps the elements in a particular order. However, the problem comes when an element needs to be extracted from the set. Lists only allow to do this if the rank of the element is known, that is, what its position in the list is.

A way to solve this, if we are fortunate enough that our framework offers disjunction as an option, would be to take either the first element, or the second element, or the third element, and so on, from the list. But apart from being cumbersome to write and use, one must know beforehand the size of the sets that might be unified.

In Borjes, Set objects allow precisely this use. There is a derivate type of object, the Direct sum, which lets the user extract a member from a Set. It is a composite, unifiable type, and has two sub-object. One of them is an element, of any Borjes type, and the other one is a remainder set, which must be a set or a variable. Direct sums are useful because of the way they unify. They only unify with Sets, and result in a number of possible alternative unifiers, represented in a disjunction.

Each unifier is the decomposition of the unifying set into an element, which unifies with the `element` of the sum, and one of the set's elements; and the `remainder` set which is the original one minus the extracted element. Direct sums are especially useful when used with variables. If the extracted element is a Borjes variable, then it can be reused somewhere else in the object to which it belongs. And the remaining set, if a variable again, can be also kept for future extraction.

Direct sums are most useful in a grammar for a free word order language, when dealing with valency. The valency may then be a set of required arguments to the syntactic element, and the parser can find (unify) them in a sentence in any order that the grammar allows.

5.3.3 Non-unifiable types

Although we have already seen some non-unifiable types, these were objects intimately related to unification. Borjes additionally provides a number of objects which do not unify, and are not part of the unification framework. Rather, these objects compose the upper layer of functionality, using the unification system to create higher level grammatical constructs.

Rules, Principles and Lexicons all represent types of linguistic concepts which form part of a grammar. The Grammar object precisely encodes this collection of objects, and is the result of grammar development. In other formalisms, a grammar would be program code, if its a computational formalism, or an abstract description in a document. Borjes grammars are just data, JSON-encoded (see before, section 5.2), and are portable and reasonably human readable.

But since they are only descriptions, another object must be used to parse text according to them. Accordingly named, the Parser object takes a grammar as configuration, and uses it to parse the text provided as input. The result is a Tree, the syntax tree of the parse.

5.3.3.1 Rules

Rules are the principal building blocks of context-free grammars. They make the derivation tree structure, by grouping leaves (words) into subtrees, then joining the nodes into larger subtrees until the top (root) is reached, hopefully finding a proper sentence according to the grammar.

```
1  var Rule = require('borjes').Rule;
2
3  var NP = Literal('NP');
4  var VP = Literal('VP');
5  var S = Literal('S');
6
7  var sv_rule = Rule (S, [NP, VP]);
8
9  Rule.apply(sv_rule, [NP, VP]); // -> S
10 Rule.apply(sv_rule, [VP, NP]); // -> Nothing
```

Listing 5.10: Example of the use of rules

Rules have a mother node, and a list of daughters. The mother and daughters are arbitrary Borjes objects, but should share the same world. If the daughters have any world bound, it is ignored, and the world of the mother is the only one used. This allows the grammar writer to specify that some information should unify across daughters, or transfer that information to the mother via variables.

When a rule is applied, a list of Borjes objects have to be passed as arguments. If these objects unify with the rule's daughters in the appropriate order, then the mother node is returned, with all variables bound to the unified values.

5.3.3.2 Principles

Principles are a kind of rule very often used in HPSG grammars. They are logical constraints that determine the proper form of an object. Objects in constraint-based formalisms, such as HPSG, use principles to better specify the objects that form a syntax tree. While in classical CFG a tree is licensed just by finding the appropriate daughter nodes (as in Listing 5.10), objects in constraint-based grammars impose a series of additional conditions on nodes to be considered correct.

An example of this might be the previously mentioned agreement principle. This principle might apply to all phrases in the grammar, and requires that all

```
1  var Principle = require('borjes').Principle;
2
3  var world = World();
4  var agr = Variable(world);
5  var antecedent = FStruct();
6  var consequent = FStruct({ head: FStruct({ gender: agr }),
7                             non_head: FStruct({ gender: agr })})
                                 ;
8  World.bind(world, consequent);
9
10 var agr_pple = Principle(antecedent, consequent);
11
12 var ok_NP = FStruct({ head: FStruct({ gender: 'feminine' }),
13                       non_head: FStruct()});
14 var bad_NP = FStruct({ head: FStruct({ gender: 'feminine' }),
15                         non_head: FStruct({gender: 'masculine'})
16                       });
17
18 Principle.apply(agr_pple, ok_NP); // -> a FStruct with a
19         // variable for the gender of both head and non_head,
20         // and that variable bound to the value 'feminine'
21 Principle.apply(agr_pple, bad_NP); // -> Nothing
22 Principle.apply(agr_pple, Literal('John')); // -> 'John'
```

Listing 5.11: Example of the use of principles

the children of the phrase must share the agreement feature of the head daughter. What this means in unification-based grammars is that every object must unify with the content assigned to it by the principles that apply.

Borjes Principles have two main components, the antecedent and the consequent. The antecedent is a kind of guard that decides when the principle applies. It is a Borjes object, which tells (via unification) what shape an object must match for the principle to be applied to it. The consequent is the actual meaning of the Principle. It is again a Borjes object, and is the one that is unified with candidate objects to constrain them. If unification fails, the result is Nothing, which is discarded by the parser.

Principle application works in a very straightforward way: before an object is considered to be definitive, after its construction (as a rule's mother, or another principle's consequent) and before its inclusion in the parse table, the antecedent is unified with it[2]. If unification fails, nothing happens, and the object is considered valid according to this principle. If unification succeeds, the result is discarded, and the object is then unified with the consequent. The result of this unification, if any, is considered valid with regard to the principle. When all principles are tested, the object can be considered definitive.

5.3.3.3 Lexicons

A Lexicon is a dictionary, a mapping from words or other surface forms to their syntactic representation. In Borjes, entries are Javascript strings, and the values are arbitrary Borjes objects. The Lexicon is used by the parser to translate input text (split into words) into Borjes objects to which then apply Rules and Principles.

Entries may be added to the dictionary one by one, by providing the index and value. More than one Borjes object can be associated with one word, to represent lexical ambiguity. When getting the entry for that word, all options will be returned.

Another way to use a Lexicon is via the "inflect" capability. This is a function which can be used to speed up development, or to actually perform morphological inflection or derivation. It is parallel to lexical rules as used in other formalisms, in that its results are pre-compiled, not computed during parsing.

First, a "morphology" function must be defined. This Javascript function must take a string, the lexeme representative, and return a list of inflected forms.

[2]As opposed to what is done in TRALE, which uses a more strict (and correct) interpretation where the object is only tested to subsume the antecedent

```
var Lexicon = require('borjes').Lexicon;

var my_dict = Lexicon();

var NN = Literal('NN');
var masc_john = FStruct({ phon: Literal('John'),
                          gender: Literal('masc') }));

Lexicon.add(my_dict, 'john', NN);
Lexicon.add(my_dict, 'john', masc_john);

Lexicon.get(my_dict, 'john'); // -> [ NN, masc_john ]

function plural (word) {
    return [
    [ word, FStruct({ lexeme: word, number: singular }) ],
    [ word+'s', FStruct({ lexeme: word, number: plural }) ]
    ];
}

Lexicon.inflect(my_dict, plural, [ 'cat', 'dog' ]);

Lexicon.get(my_dict, 'cat'); // -> { 'cat', 'singular' }
Lexicon.get(my_dict, 'cats'); // -> { 'cat', 'plural' }
Lexicon.get(my_dict, 'dogs'); // -> { 'dog', 'plural' }
```

Listing 5.12: Example of the use of the lexicon

Each inflected form is a pair, the first element being the actual entry to be used for the Lexicon, the second one an associated value. If more than one pair has the same first value, all the second values will be associated to it, and all will be returned from the Lexicon when querying for the key.

The modifications made by the morphology function to the lexeme representative are arbitrary, and can be any the user desires so long as the result is again a string. For agglutinative morphology, suffices may be concatenated. For fusional phenomena, or for irregular forms, the string might be more or less transformed. The characters may be even translated into a different alphabet, for a cross-lingual lexicon.

5.3.3.4 Grammars

We have now enough elements to construct a formal grammar for a language. The `Grammar` object just puts them all together, in a single container object. This object is then used by the Parser to build syntactic trees out of plain text.

5.3.4 Parsers

The parser object is the main execution-time piece of functionality in Borjes. Once the Grammar object has been populated by the grammarian, using the objects from the previous steps, it is time to test or use it on actual text. The Parser keeps some internal state, including the grammar and previously parsed input. The user can feed words sequentially to the Parser, and for each word a parse chart is filled in. Objects spanning different lengths of input will then be stored there.

The Parser is built with one single argument, the Grammar to be used. It then exposes three methods:

- **input**. This methods receives one single word, and fills in all the corresponding new cells in the chart table. It provides the basic functionality of the Parser.
- **reset**. All previous input is forgotten, and the chart is cleaned.
- **parse**. For convenience, a method called `parse` is provided. This method resets the parser, and then reads in all words in the input sentence. It also returns the top of the parse chart, so it provides an encapsulation of all the Parser's functionality.

Note that the Parser receives the input partly preprocessed, segmented into words. The segmentation must be the same that was used for building the Lexicon, and is left to the user. Simple space splitting, removal of punctuation, or more sophisticated morpheme analysis may be performed, so long as the "words" that the Parser receives are the same as those that it will find in the Lexicon.

The objects stored in the parse chart, and thus, the result returned by the Parser, are in the form of Trees (see subsection 5.3.5). These trees have as their root the top-level object, and their children are the mid-level objects that the root was parsed from. The leaves of the Tree will therefore be the words that

```
1 // continued from previous examples
2
3 var Grammar = require('borjes').Grammar;
4
5 var my_grammar = Grammar([sv_rule, vo_rule, np_rule],
6                           my_dict,
7                           [agr_pple, other_pple]);
```

Listing 5.13: Example of the instantiation of a grammar object

```
var Parser = require('borjes').Parser;

var my_parser = Parser(english_grammar);

Parser.parse(my_parser, ['John', 'loves', 'Mary']);
// -> Tree(S, ...)

Parser.parse(my_parser, 'dearly');
// -> Tree(Adv) (it has been reset)

Parser.parse(my_parser, ['John', 'Mary', 'Bob']);
// -> Nothing
```

Listing 5.14: Example of the use of the parser

were input to the Parser. If the Grammar stores the derivation structure in the grammar objects themselves, the returned tree can be discarded and only the root kept. It is useful, however, to inspect the tree for debugging or instructional analysis.

Parsing works, as mentioned, via the standard bottom-up, breadth-first chart parsing algorithm. For now it is only binary, but the code structure allows it to parse rules with more than two children too. The algorithm works by exhausting bottom-level parse chart cells, before moving up a level. For each cell, all possible children are found (called "legs" in the implementation), and then all Rules (subsubsection 5.3.3.1) are tried on them. Every result is then inspected according to all of the Principles (subsubsection 5.3.3.2), and if it is found to be valid, it is inserted in the corresponding cell.

Every time a new input is given, a new bottom-most cell is added. The algorithm works upwards, exhausting all cells, that cover more previous input each time. At the top of the chart, a cell spanning all input is found, and the objects in that cell are considered to be the results of the parse.

5.3.5 Trees

Trees represent classical, simple tree data structures. They have a `node` property, which holds the object in the current node, and a `children` array, with all the children of the node. The children can be any Borjes objects, but if they themselves are Trees, then an arbitrary-length, recursive Tree data structure can be constructed, and that is what the Parser returns.

Trees provide a special functionality, by which they can be converted into s-expressions (McCarthy 1960), not unlike the ones used in Lisp. In Javascript, they are encoded in arrays instead of parenthesis, because of differences in the syntax

```
var Tree = require('borjes').Parser;

var my_tree = Tree('S', Tree('NP', 'John'),
                        Tree('VP', Tree('V', 'loves'),
                                   Tree('NP', 'Mary')));

Tree.to_sexp(my_tree); // -> ['S', ['NP', 'John'],
                       //          ['VP', ['V', 'loves'],
                       //                 ['NP', 'Mary']]]
```

Listing 5.15: Example of the use of trees

of the programming languages. The first element of the s-exp is the converted root of the Tree, and the rest of the elements are the converted children. If the children are Trees, then their conversion will be itself a s-exp.

The default conversion takes the Borjes object in the node and puts it into the s-exp corresponding place, but the user is allowed to provide a custom conversion function which further processes the Borjes object and transforms it in any desired way. This is used by the Treebanking module in BOLDE, and can be useful for quick inspection of large trees.

5.4 Using the library

In the previous section, we have explored the objects that Borjes exports. Along with them, code listings have been provided, to get a feel of the general use of the library. A few functions remain to be explained, and they are listed here.

5.4.1 Unify

The unify function has appeared in code listings of most unifiable types, so it hardly requires introduction. It can be imported directly from the Borjes module with `require('borjes').unify`. It takes two Borjes objects, and returns their most general unifier (or unifiers, if there are multiple possibilities) as a Borjes object. This unifier might be Nothing, which signifies that there is no such unifier.

A third parameter to `unify` is an optional boolean 'as_array'. If true, the most general unifiers are always returned as an array, empty if there was no unifier, or with the different unifiers as the array elements. If 'as_array' is false (the default), if no unifier was found, Nothing is returned; if there was only one possible unifier, it is returned; and if there were multiple options, they are returned as an array as if 'as_array' was true.

5.4.2 Copy

Another useful function is provided with the type system in Borjes. It is imported from the 'types' module, with `require('borjes').types.copy`. It can be used when one needs an object that is identical in content to another one, but not actually the same.

If the same object was reused, then further modifications to any of them would alter the other one too. Of course, this is not a problem for constant (primitive) types, which are passed as-is by the copy function. It is good practice, nonetheless, to still use the copy function everywhere, to make a consistent use of the library.

The copy function is used pervasively in the unification module, and it accepts an optional second argument, a `map` which can transform variables between worlds. This parameter should be considered an implementation detail, and treated as not present by the library user.

5.4.3 Normalize

When many unifications happen to an object, and variables are present in this unifications, their number might grow to be unmanageable. Unification never destroys any variables, but can create them for sharing structure. Sometimes an object may end up having a sub-object hidden under a variable, but this variable is not referenced anywhere else. Another common problem is variables that are bound to other variables, making a chain of bindings which is cumbersome and difficult to understand.

A normalization function is provided with the type module, required with `require('borjes').types.normalize`. It takes a Borjes object and returns an equivalent one, in which all redundant uses of variables have been removed. It should only be used with objects which are considered "final", since it may destroy some bindings which are meaningful to an ongoing unification process. The Parser calls `normalize` every time before inserting an object in a cell, so all results of the parsing process are already normalized.

5.4.4 Comparison

A final utility function is provided with the type module. It is found under `require('borjes').types.compare`, and takes two Borjes objects and returns a boolean value. The return value is true if the objects where found to be similar, false otherwise. It is not a very intelligent function, and should not be relied upon. Borjes objects are either equal or not, and whether they are equivalent

```
var Grammar = require('borjes').Grammar;

var my_grammar = Grammar.CFG('textual description of the
    grammar');
```

Listing 5.16: Example of the use of the yaml loader

to the user might have more nuances than just whether the same properties are present with comparable values. This function is mostly used for testing the unification module, and should not be necessary for the library user.

5.4.5 YAML Grammars

There is another piece of functionality that Borjes exports. This one is exported as a method of the Grammar object, but is implemented as its own module. It is a reader, which takes a textual description of a CFG grammar in YAML format (Ben-Kiki, Evans, and Ingerson 2009), and creates a Borjes grammar. This textual format, however, does not support most of the Borjes type system, so it is only useful for grammars which do not require the power of the unification framework. It is used as per Listing 5.16. In section 6.1 an example of such a grammar is given.

5.4.6 Packaging and install

Borjes is written as a CommonJS module (*CommonJS module specification* 2015), and distributed as an NPM package (*NPM - the node package manager* 2015). It is not listed as a public package yet, but there are plans to publish it. For now, the latest version can be found at the url `https://garciasevilla.com/pkgs/borjes.tar.gz`. It works in both server environments (testing is performed with node.js) and modern browsers, which support the ES5 standard. Read access to the git repository for the code can also be provided upon request.

5.4.7 Structure of the code

Code is distributed in a series of files, located in the `src` directory. The `test` directory contains the unit tests of the library, and the `lib` directory provides a modified version of the *lisp2js*[3] library. Only the `test` and `src` directories are distributed with the npm package. Source code lives in a private git repository to which read access can be obtained by contacting the author.

[3] `https://github.com/shd101wyy/lisp2js`

The source files in the `src` directory are as follows:

- **cfg.js**. This file contains the code for reading extended CFG grammars from a YAML description (subsection 5.4.5).
- **formatter.js**. This module exports two functions for printing a few Borjes objects as strings. It is a legacy module and is deprecated.
- **grammar.js**. The Grammar object is defined here (subsubsection 5.3.3.4).
- **index.js**. This file is the main entry point of the library, sourced by the package manager when requiring Borjes.
- **lexicon.js**. The source for the Lexicon object (subsubsection 5.3.3.3).
- **parenthesis.js**. This module provides functionality for parsing parenthetical s-expressions. It is used in the CFG reader and by the Treebanking module of BOLDE.
- **parser.js**. The Parser object is defined here (subsection 5.3.4).
- **principle.js**. Source code for the Principle object (subsubsection 5.3.3.2).
- **rule.js**. This file describes the Rule object (subsubsection 5.3.3.1).
- **tree.js**. The Tree object is defined here (subsection 5.3.5).
- **types.js**. The main source file, contains the definitions of the unifiable and related Borjes objects, and the utility functions for dealing with them (subsection 5.3.2, section 5.4).
- **unify.js**. Unification is implemented in this module. A single function is exported, the rest should be considered implementation detail and only for the use of the exported function (subsection 5.3.1, subsection 5.4.1).

5.5 Borjes-react

Alongside with Borjes, another Javascript library has been developed. This is **borjes-react**, which is also distributed as an npm package, at the url `https://garciasevilla.com/pkgs/borjes-react.tar.gz`. The code is also stored in a git repository to which access can be granted.

Borjes-react is a bridging library, which exports a React component (see subsubsection 4.3.2.3). This component takes as parameter a borjes object, and

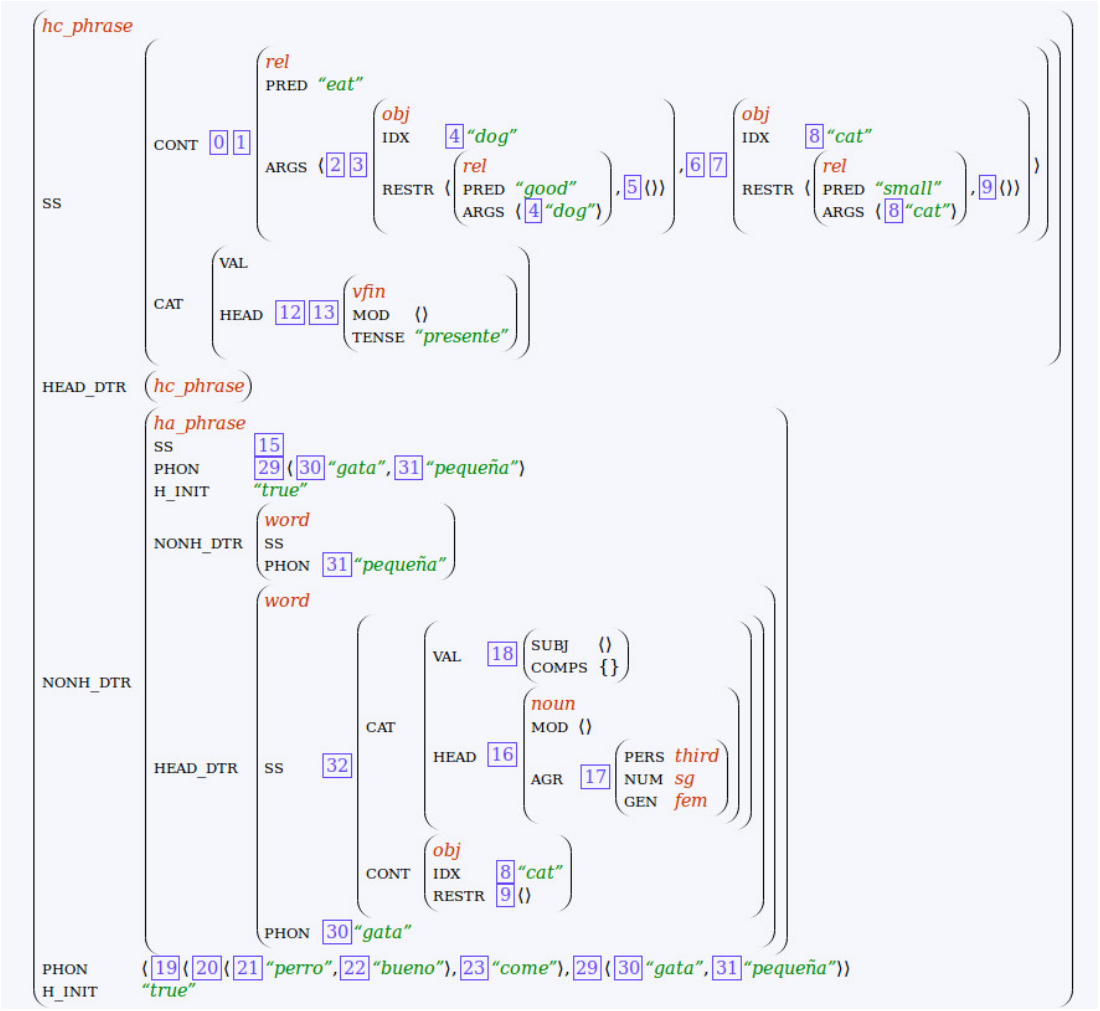


Figure 5.3: A big feature structure in borjes-react

displays it visually in HTML, as can be seen in Figure 5.3. It also allows the component to be edited graphically (Figure 5.4), and this functionality is the basis for the Visual Editor in BOLDE (subsection 4.3.3). It can be used independently, too, as Listing 5.17 shows.

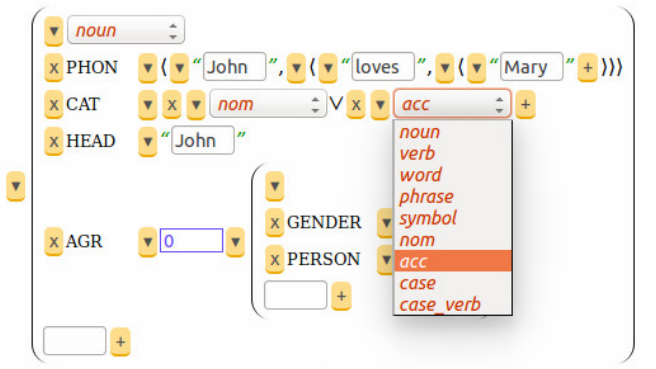


Figure 5.4: Editing in borjes-react

```
var React = require('react');
var Borjes = require('borjes');
var BorjesComponent = require('borjes-react');

var editable = false; // If true, the component can be edited
var cpbuffer = {}; // A buffer for copy-pasting borjes objects
var L = Borjes.types.Lattice(); // The lattice to use for
// the type of feature structures (if any).

// this example uses JSX syntax, so must be precompiled to
// Javascript, for example with react-tools or babel.
function render (value) {
    React.render(<BorjesComponent
                x={value}
                update={render}
                cpbuffer={cpbuffer}
                opts={{editable, signature:L}}
    />, document.getElementById('body'));
}

var fstruct = Borjes.types.FStruct();
var world = Borjes.types.World();
Borjes.types.World.bind(world, fstruct);

render(fstruct);
```

Listing 5.17: Example of the use of Borjes-react

5.6 Summary

With Borjes, the software developed for this master thesis is complete. Borjes provides the low-level details, the unseen engine that allows BOLDE to offer the user the required linguistic power. It does so by providing software tools to create and use formal grammars within the unification-based paradigm.

In the next chapter, the results of putting BOLDE and Borjes together, with two examples of their use, are described. The original aims and objectives from chapter 1 are then evaluated, and in chapter 7 the final conclusions are drawn, and future work is outlined.

Chapter 6

Results

For the purpose of evaluating the system, two different computational linguistic projects have been developed in it. They can be accessed online, at `https://garciasevilla.com/bolde`, available with the user name `thesis` and the password `evaluationexamples`. This user is automatically reset every day, so modifications or other destructive tests are not a problem.

6.1 Use case 1: A numerical, lexicalized CFG grammar

One of our main aims from section 1.4 was to have a linguistically oriented system. BOLDE, with Borjes, is able to parse text according to many different grammars. In this example, a simple context free grammar is discussed. It is then extended, to show the flexibility of the system. It also demonstrates the configurability of the execution pipeline, and the treebanking component. It can be found in the online installation under the project named "CFG".

6.1.1 The base grammar

This first grammar will be in YAML data exchange format. This format is easy to write and simple enough for it to be adapted for a CFG grammar. The code is provided in Listing 6.1, and is in the file `english.yml` on the server.

This grammar has a number of simple rules which state that when the children nodes (the different symbols) are found, the mother node can be constructed. The YAML format supports different options for the children of a rule, as can be seen in the example in Listing 6.1. The lexicon is similar, but the children of a rule

```yaml
%YAML 1.2
--- # Simple English Grammar
Rules:
    S:
    - NP VP
    - S ConjS
    NP:
    - Det NP
    - Adj NP
    - NP ToVP
    - NP PP
    - NP REL
    VP:
    - Adv VP
    - VP Adv
    - VP NP
    - VP Adj
    - VP PP
    PP:
    - Prep NP
    REL:
    - That VP
    ConjS:
    - Conj S
    ToVP:
    - To VP
Lexicon:
    NP: [ John, Mary, dog, store, lettuce, salad, program, code,
        student, professor, he ]
    VP: [ loves, likes, walks, runs, drives, buys, make, writes,
        works, comments, enjoys ]
    Conj: [ and ]
    To: [ to ]
    That: [ that ]
    Det: [ the, a, his ]
    Adj: [ brown, grocery, short, good, "John's" ]
    Adv: [ deeply, quickly, correctly ]
    Prep: [ to, for, like ]
```

Listing 6.1: CFG grammar for English

```
John deeply loves Mary
John likes Mary
John likes the brown dog
John walks to the store
John walks to the grocery store
John runs to the grocery store
John quickly runs to the grocery store
John drives to the grocery store
John drives to the grocery store and he buys lettuce
John drives to the grocery store and he buys lettuce to make a
   salad
John drives to the grocery store and he buys lettuce to make a
   salad for Mary
John writes a short program
John writes a short program that works correctly
John writes a short program that works correctly and he
   comments his code like a good student
```

Listing 6.2: Test sentences for the CFG grammars

are the terminal symbols of the grammar, and the mother node is the assigned non-terminal.

The format used by Borjes is a bit different from what one would expect in usual CFG grammars, but is still in Chomsky normal form (Chomsky 1959), so it is enough for expressing any such grammar.

Now we want to parse some sentences according to this grammar. The test sentences are in the file `test.txt`, whose contents are in Listing 6.2. Using the grammar to parse these sentences gives results such as those in Figure 6.1 and Figure 6.2.

To obtain these results it is necessary to click on the "Run" button, the right-pointing black arrow on the top right of the project tab. But first, a `run.yml` file must be written, to tell the system how to connect the different components. The basic file needed for running the test above can be found in Listing 6.3.

As we can see, the system is able to parse arbitrary CFG grammars in CNF. It is also able to display the parse trees visually, showing the derivation tree and the different symbols in a graphical way. The different branches of the tree can be shown or hidden by clicking on the lines, for easier inspection. We cannot see it in this document, but editing is real-time. If a user edits the `english.yml` file, any other user who is simultaneously viewing the same file will be able to see the edits. He can respond with other changes, and the first user will immediately see them. The new contents are also automatically saved on the server.

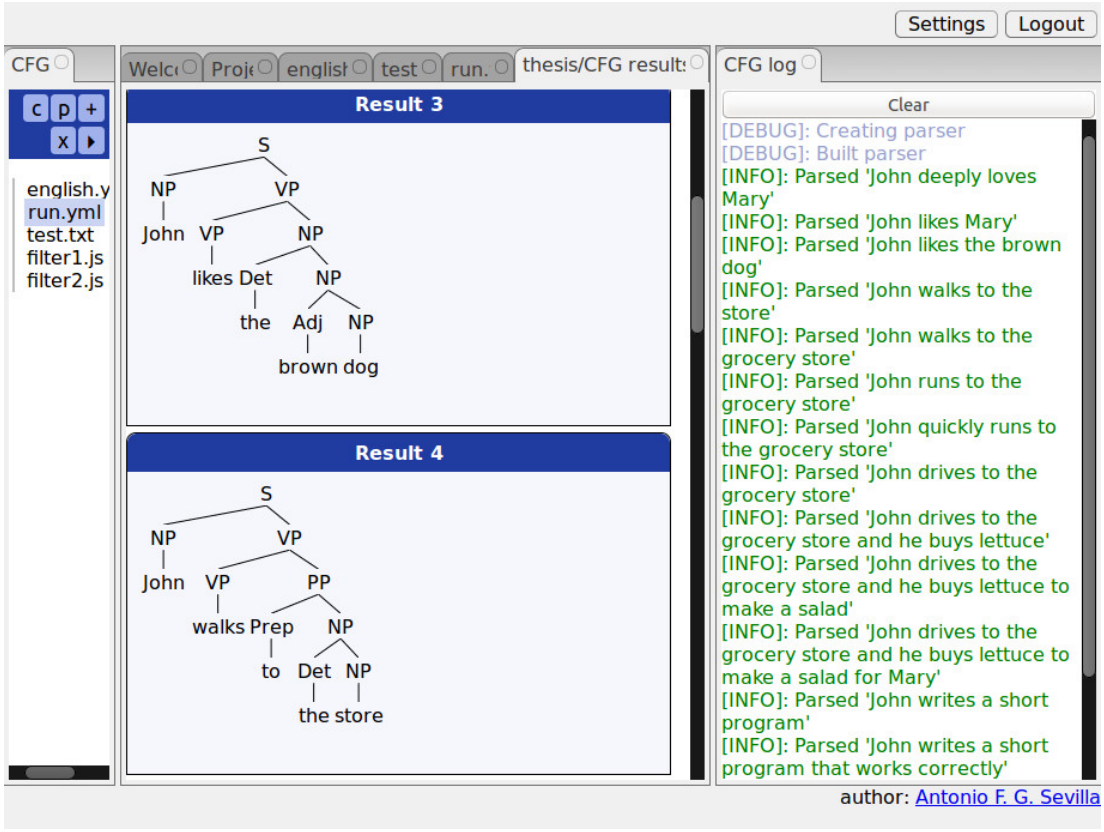


Figure 6.1: Screenshot of the parse results

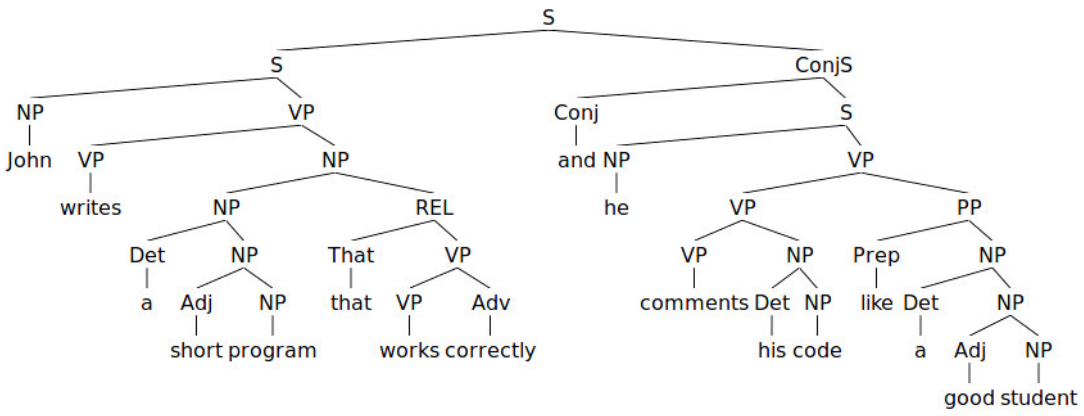


Figure 6.2: A parse tree for a long sentence

```yaml
elements:
- &source
  type: filesource
  files:
  - test.txt
- &parser
  type: borjes
  config:
    format: YAML
  files:
    grammar: english.yml
- &display
  type: display
connect:
- [ *source, *parser, *display ]
```

Listing 6.3: Simple run.yml file

6.1.2 Lexicalized

But the system is not limited to this type of grammar. If we remember John from the introduction (section 1.3), he wanted to be able to modify how the system worked and specify new functionality. There are a number of ways in which this can be done.

The YAML grammar format is not restricted to classical CFG grammars. While it does not have the full expressivity of the Borjes engine (chapter 5), it gives access to a simple extension: annotated nodes. For example, in Listing 6.4 the grammar has been extended so that nodes include also the head of the phrase, a lexical annotation. This is done by using parenthetical expressions, which can be added to a symbol to express that the node is annotated with a list of variables. These variables must have been found in the children, or their value will be undefined. In the lexicon, variable 0 means the terminal string, and 1, 2, 3... any possible annotations it may have (see next subsection for this). Figure 6.3 shows the results of using the LCFG grammar in BOLDE. To test it online, substitute `lcfg.yml` for the parser in the `run.yml` file.

6.1.3 Numerical

And how can we use this extensibility to help John? Using the new annotation feature, we can encode probabilities into the terminal nodes, or any other number that might express a numerical hypothesis. Then, we just need one other tool, that of predicates.

Predicates allow the grammar builder to transform the values attached to

```
%YAML 1.2
--- # Simple English Grammar
Rules:
    S(lex):
    - NP VP(lex)
    - S(lex) ConjS
    NP(lex):
    - Det NP(lex)
    - Adj NP(lex)
    - NP(lex) ToVP
    - NP(lex) PP
    - NP(lex) REL
    VP(lex):
    - Adv VP(lex)
    - VP(lex) Adv
    - VP(lex) NP
    - VP(lex) Adj
    - VP(lex) PP
    PP(lex):
    - Prep NP(lex)
    REL(lex):
    - That VP(lex)
    ConjS(lex):
    - Conj S(lex)
    ToVP(lex):
    - To VP(lex)
Lexicon:
    NP(0): [ John, Mary, dog, store, lettuce, salad, program, code
        , student, professor, he ]
    VP(0): [ loves, likes, walks, runs, drives, buys, make, writes
        , works, comments, enjoys ]
    Conj(0): [ and ]
    To(0): [ to ]
    That(0): [ that ]
    Det(0): [ the, a, his ]
    Adj(0): [ brown, grocery, short, good, "John's" ]
    Adv(0): [ deeply, quickly, correctly ]
    Prep(0): [ to, for, like ]
```

Listing 6.4: Lexicalized CFG grammar

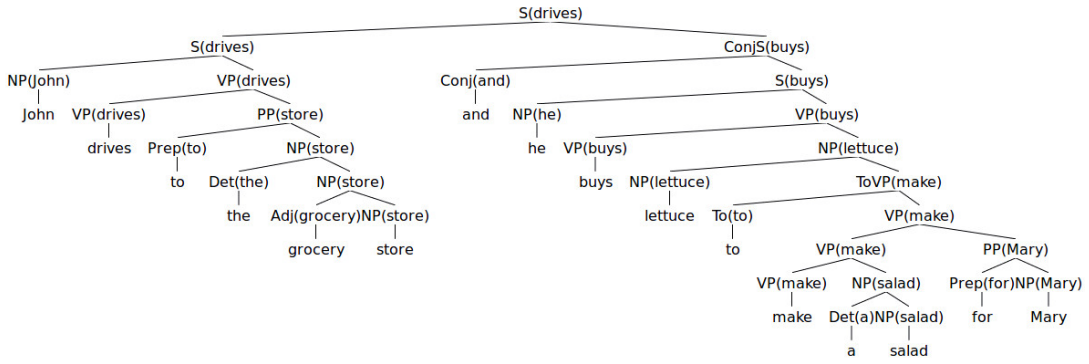


Figure 6.3: A parse tree of LCFG

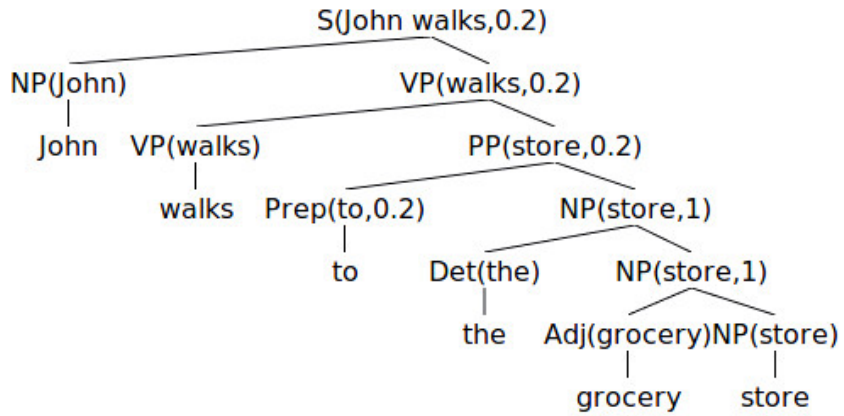


Figure 6.4: A PCFG tree

a node as annotation, by using any arbitrary code. This code is stated in a `Predicates` section in the grammar file, and can then be used in the annotation list of any rule mother. An example, in which a predicate in Lisp and another in Javascript are used, is given in Listing 6.5. There, nodes are annotated with numerical values, and then used (multiplied) by the parser when building the derivational tree. This grammar is also online, with the name `pcfg.yml`, and can again be used by putting it in the pipeline in the `run.yml` file. A result from its parse can be seen in Figure 6.4.

6.1.4 Pipeline configuration

As we have seen, the execution pipeline is flexible, allowing us to change what grammar to use depending on what we want to test. But its flexibility is even greater. The `run.yml` file lets us set up an almost arbitrary data flow, by connecting different components in different ways. An example pipeline can be found in Listing 6.6. It is also explained as a diagram in Figure 6.5.

The elements section is optional, instantiating components that can later be used thanks to YAML's anchor feature. In the connect section, a series of lines can be listed. Each line's elements are connected, the output of each connected to the input of the next one. Tee elements make the data flow merge or diverge, by connecting all their inputs to all their outputs.

6.1.5 Code extensions

Another way that system execution is extensible is with code extensions. These are files in the project, with arbitrary code, that can be placed anywhere in the

```
%YAML 1.2
--- # Simple English Grammar
Predicates:
    Prod: !lisp |
        (fn (:x 1 :y 1 :z 1)
            (* x y z))
    Concat: !js/function |
        return arguments[0]+' '+arguments[1];

Rules:
    S(Concat(lex1,lex2),Prod(prob1,prob2)):
    - NP(lex1,prob2) VP(lex2,prob1)
    - S(lex1,prob1) ConjS(lex2,prob2)

    NP(lex,Prod(prob1,prob2)):
    - Det(_,prob1) NP(lex,prob2)
    - Adj(_,prob1) NP(lex,prob2)
    - NP(lex,prob2) ToVP(_,prob1)
    - NP(lex,prob2) PP(_,prob1)
    - NP(lex,prob2) REL(_,prob1)

    VP(lex,Prod(prob1,prob2)):
    - Adv(_,prob1) VP(lex,prob2)
    - VP(lex,prob2) Adv(_,prob1)
    - VP(lex,prob2) NP(_,prob1)
    - VP(lex,prob2) Adj(_,prob1)
    - VP(lex,prob2) PP(_,prob1)

    PP(lex,Prod(prob1,prob2)):
    - Prep(_,prob1) NP(lex,prob2)

    REL(lex,Prod(prob1,prob2)):
    - That(_,prob1) VP(lex,prob2)

    ConjS(lex,Prod(prob1,prob2)):
    - Conj(_,prob1) S(lex,prob2)

    ToVP(lex,Prod(prob1,prob2)):
    - To(_,prob1) VP(lex,prob2)

Lexicon:
    NP(0): [ John, Mary, dog, store, lettuce, salad, program, code
        , student, professor, he ]
    VP(0): [ loves, likes, walks, runs, drives, buys, make, writes
        , works, comments, enjoys ]
    Conj(0): [ and ]
    To(0,1): [ to(0.8) ]
    That(0): [ that ]
    Det(0): [ the, a, his ]
    Adj(0): [ brown, grocery, short, good, "John's" ]
    Adv(0): [ deeply, quickly, correctly ]
    Prep(0,1): [ to(0.2), for, like ]
```

Listing 6.5: Numerical CFG grammar with predicates

```yaml
elements:
- &source
  type: filesource
  files:
  - test.txt
- &english
  type: borjes
  config:
    format: YAML
  files:
    grammar: english.yml
- &display
  type: display
- &lcfg
  type: borjes
  config:
    format: YAML
  files:
    grammar: lcfg.yml
- &pcfg
  type: borjes
  config:
    format: YAML
  files:
    grammar: pcfg.yml
- &filter1
  type: javascript
  files:
  - filter1.js
- &filter2
  type: javascript
  files:
  - filter2.js
- &treebank
  type: treebank
  files:
    - result
- &teeout
  type: tee
- &teein
  type: tee
connect:
- [ *source, *teeout ]
- [ *teeout, *filter1, *english, *teein ]
- [ *teeout, *filter2, *english, *teein ]
- [ *teein, *display ]
- [ *teein, *treebank ]
```

Listing 6.6: A complex pipeline

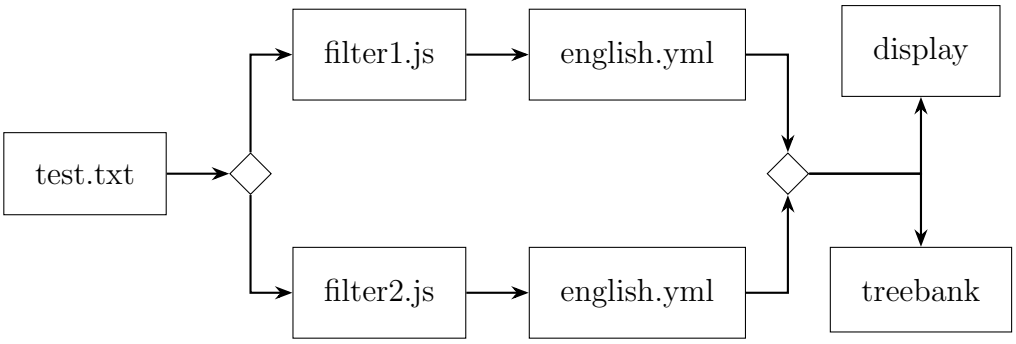


Figure 6.5: A diagram of the pipeline in Listing 6.6

pipeline. In the previous section, files `filter1.js` and `filter2.js` are examples of this. They are written in javascript, and can perform any actions that the user might need, even modify the outputs from the parser stage, or annotate them with extra information. They could also connect to the internet, and download data from third party sources to include in the pipeline, or maybe send data to another application. In the example project, these are really simple functions which only filter some of the input sentences. For a more sophisticated example see subsection 6.2.5.

6.1.6 Treebank

Another new component in the previous pipeline is the treebank. This is an experimental component and offers very little functionality. When a pipeline's data goes into a treebank, all the trees in the flow are stored in a temporary database in the user's computer. This database is later accessible for searching. The search is performed using a form of s-expressions, which must match the symbols in the tree for it to be returned.

For example, we can direct the final output from the first CFG grammar to the treebank, by changing 'display' to 'treebank' in the `run.yml` file. If we then search for `VP(VP(VP(VP)))`, we will find any nodes with a VP embedded four times to the left. There is only one such tree in the treebank, as we can see in Figure 6.6. We can search by more than one child, as with `NP(NP,ToVP)`, to get all NP's with a 'to + VP' phrase attached. An underscore (`_`) is an anonymous variable, allowing us to search for VP's with a right child which is also a VP, by using `VP(_,VP)`. The results can be seen in Figure 6.7.

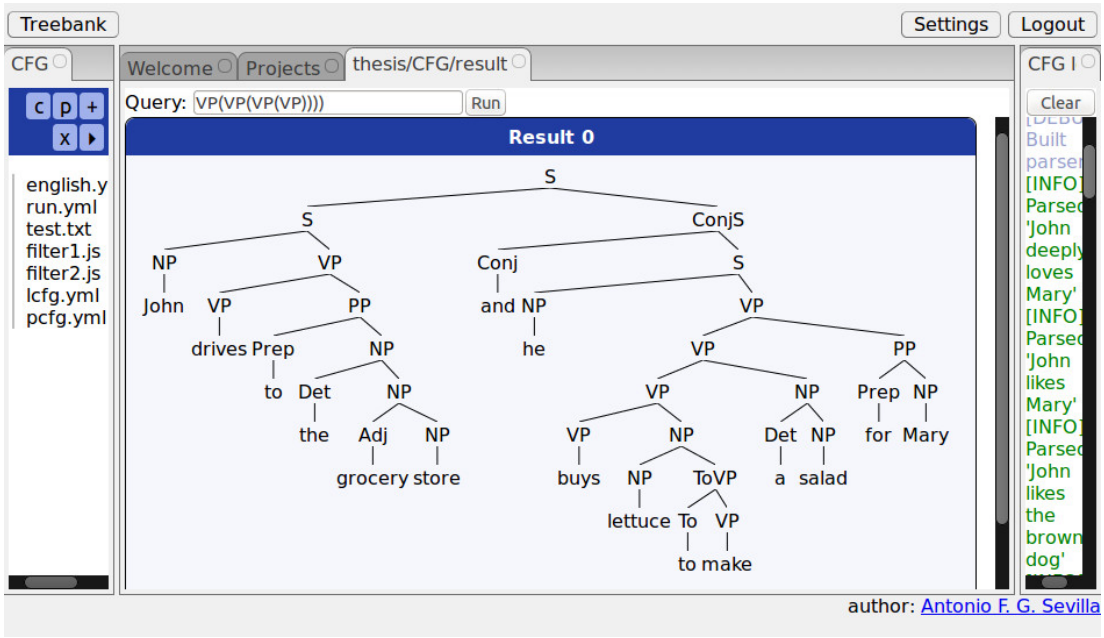


Figure 6.6: Treebank searching results for VP(VP(VP(VP)))

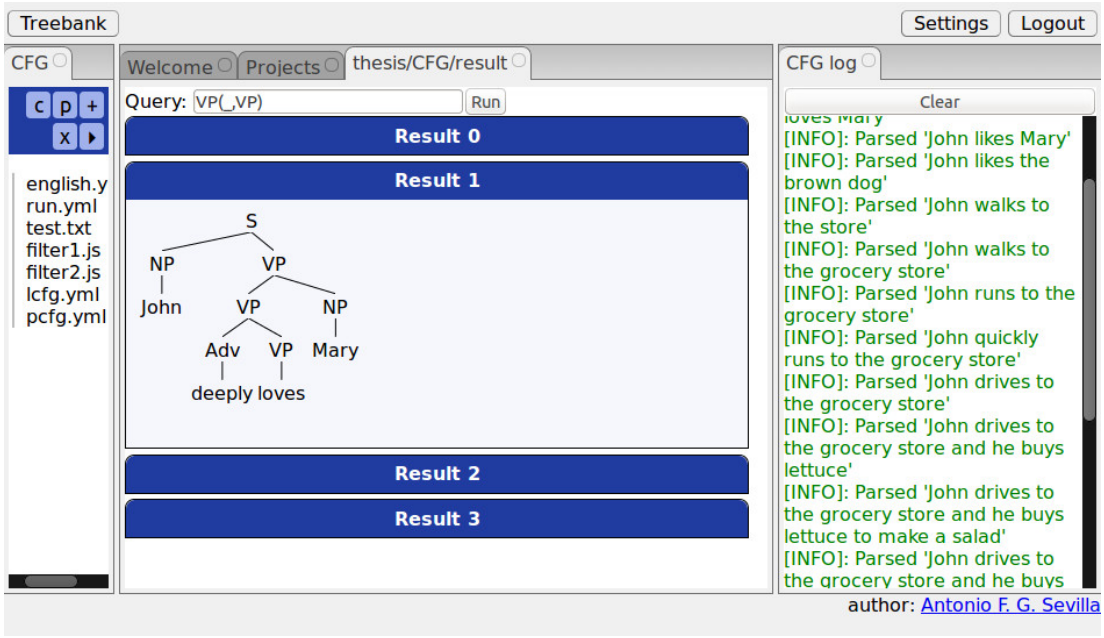


Figure 6.7: Treebank searching results for VP(_,VP)

6.2 Use case 2: An example HPSG grammar of Spanish

For our second example, we will use a grammatical formalism more powerful than that of CFG grammars. Even with the extensions of annotated nodes and predicates, the YAML format for CNF grammars is severely limited in its expressive

power. This is one of the reasons why unification-based grammars exist, which still use phrase structure rules, but do so in a more flexible and expressive way.

With the Borjes linguistic engine underneath, BOLDE is capable of using these kinds of grammar. Users can create and edit them, and including them in the pipeline just as before, obtain the parse trees and structures for test corpora. As explained in chapter 4, these grammars are edited visually in BOLDE, letting the user encode their linguistic knowledge in a way that is not only graphical and intuitive, but also closer to the way the theories are usually theoretically formulated.

For demonstrating this graphical editing of sophisticated grammars in BOLDE, a simple HPSG grammar of Spanish has been developed, and is available online, alongside the previous example, in the project named "HPSG". It is mostly based on examples from the course Linguistic Theories and Grammar Formalisms by Alexandr Rosen at Charles University in Prague.

6.2.1 The signature

While Borjes does not need well-typedness in the signature, in HPSG it is customary to use it (but not necessary, either). It serves, however, as good documentation of the structure of the objects in the grammar, and so the signature that will be presented here will also have some annotation in this regard. The full structure can be seen in Figure 6.8.

The basic type in the signature is the `sign`, displayed in Figure 6.9. Signs represent the units of the parse tree, and have a `phon` feature for the surface form (in most cases not actual phonetics, but rather the string representation of the sign, its spelling). They also have a synsem structure, abbreviated `ss`, which holds the syntactic (`cat`) and semantic (`cont`) features.

The `cat` structure has a `head` feature, which holds the syntactic content of a word, or of the head of the phrase. In traditional Spanish syntax, where phrases are called *sintagmas*, the head would be called *núcleo*. The `cat` also has the `val` structure, short for valency, which has the `subj` and `comps` features, which respectively specify the subject and complements a word might need to be saturated, or to achieve its maximal projection.

The `sign` type has two more specific subtypes, those of `word` and `phrase`. The `phrase` has three new features, a `head_dtr` (head daughter, the daughter of the phrase which supplies the head) and the `nonh_dtr` (the non-head daughter). According to whether the non-head daughter is a complement or an adjunct to the head, the two subtypes `hc_phrase` and `ha_phrase` are defined. The `phrase`

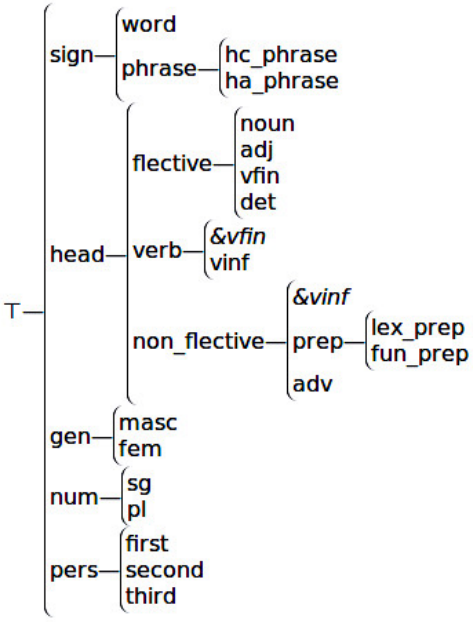


Figure 6.8: The full signature for the grammar

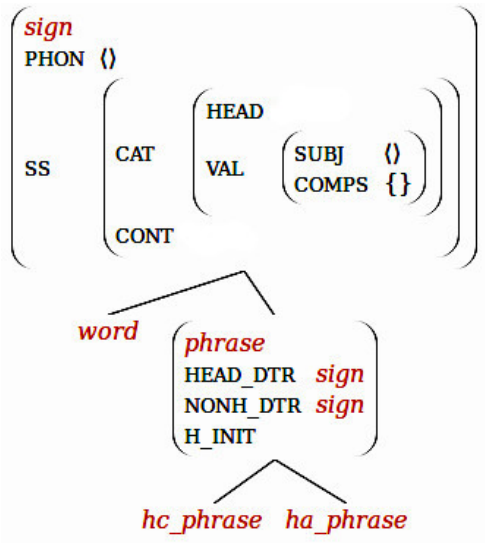


Figure 6.9: Hierarchy of signs

type also brings a new feature, `h_init`, which specifies whether the head is the left or right daughter of the phrase.

Another important hierarchy is that of `head` values, which can be seen in Figure 6.10. These are the usual syntactic categories, like noun or verb, and all share a `mod` feature. This feature is of use to modifier words for specifying the syntactic heads that they act as modifiers of. According to whether they change form to reflect grammatical categories like gender or number, heads are divided into `flective` and `nonflective`. Flective words have an `agr` feature, for `gen` (gender), `num` (number) and `pers` (person) features. These are usually called *accidentes gramaticales* in Spanish. Pure flective heads are `noun`, `adj` and `det`. Determiners also have a `dform` feature that specifies the type of determiner.

A special case among the head types are `verb`s, since they have both flective and nonflective forms, `vfin` and `vinf`. Finite verb forms (`vfin`) have a `tense` feature, which in Spanish reflects the time of the action and often other aspects like conditionality or continuity.

Apart from infinitive verbs (`vinf`), other nonflective head types are `adv` (adverbs), and `prep` (prepositions). Prepositions have a `pform` feature, which just specifies what the preposition is. In Spanish there are no cases, and prepositions often act as substitutes for marking the different complements to a verb. Often, a verb will require a prepositional phrase in its valency, but only one with the appropriate `pform`. For these instances where prepositions are mere functional

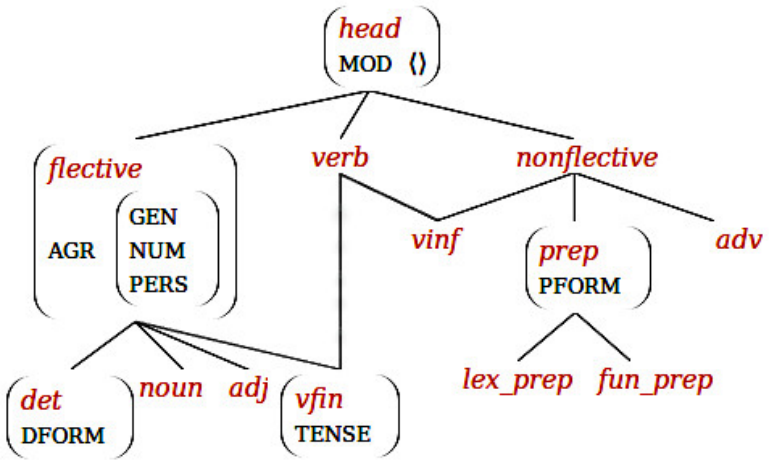


Figure 6.10: The hierarchy of head types

markers, the `prep` type is specialized by `fun_prep`. When the preposition actually does have some independent meaning, it is classified as `lex_prep`.

A final word is also needed on the semantic representation. It is a custom format, loosely inspired by minimal recursion semantics (Copestake et al. 2005). Every semantic object (the `cont` feature of the `sign`) is a feature structure with a `pred`, which specifies the "name" of a predicate. Then there are some `args`, a list of the arguments to the predicate. And finally, the `restr`, a list of restrictions, which impose further semantic constraints on the predicate. Nominal objects in this grammar are just zero-argument predicates.

6.2.2 The constraints

Now, the different signs from the signature have to be put together, into rules and principles that govern how a sentence will be parsed.

There are two rules, each coming in two versions. They look very similar, and only change in the type of the phrase they license. One of the rules is for joining a head with a complement, `hc_phrase`, the other for joining it with an adjunct `ha_phrase`. The rules also capture and concatenate the `phon` feature of the daughters. The two versions are according to whether the head daughter is the left or the right one. They can be seen in Figure 6.11.

The Principles are the constraints in charge of shaping up the phrases, and enforcing the different types of linguistic relations in the grammar. They are listed below, and some can be seen in Figure 6.12.

- **Head feature**. This principle makes sure that the HEAD feature of the head daughter is the same as the HEAD feature of the phrase.

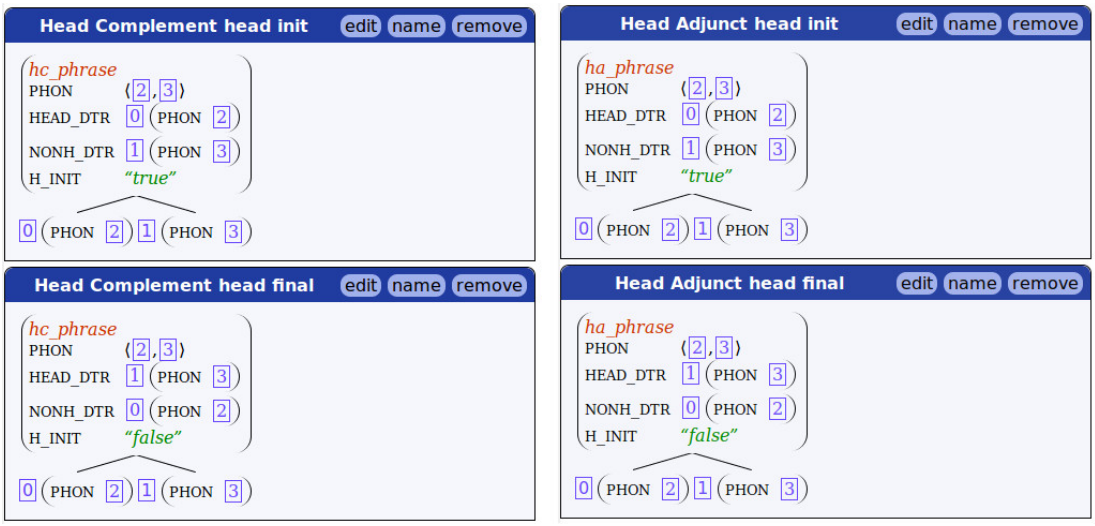


Figure 6.11: The four versions of the rules

- **Complement valency**. This principle is in charge of constraining head complement phrases, by specifying that the non-head daughter must be an object found in the valency of the head daughter. The valency of the phrase is then the same as that of the head daughter, save for the element which was found. This principle can mean finding either the subject or another complement, so it makes use of a disjunction.

- **Adjunct valency**. In a head adjunct phrase, the valency of the phrase is that of the head daughter.

- **Adjunct modifier**. Similar to the valency in head-complement phrases, in head-adjunct ones what is found is the element of the non head daughter's MOD feature.

- **Complement modifier**. In a head-complement phrase, however, the MOD feature of the phrase is empty.

- **Complement semantics**. When a head finds a complement, it is the semantics of the head that determines the relation, and are thus propagated to the phrase.

- **Adjunct semantics**. On the other hand, adjuncts are the ones that specify the semantics of the phrase, despite being the non-head daughters, so the CONT feature of the phrase is the same as that of the adjunct.

- **Preposition before NP**. This feature specifies that in prepositional phrases the preposition must come first.

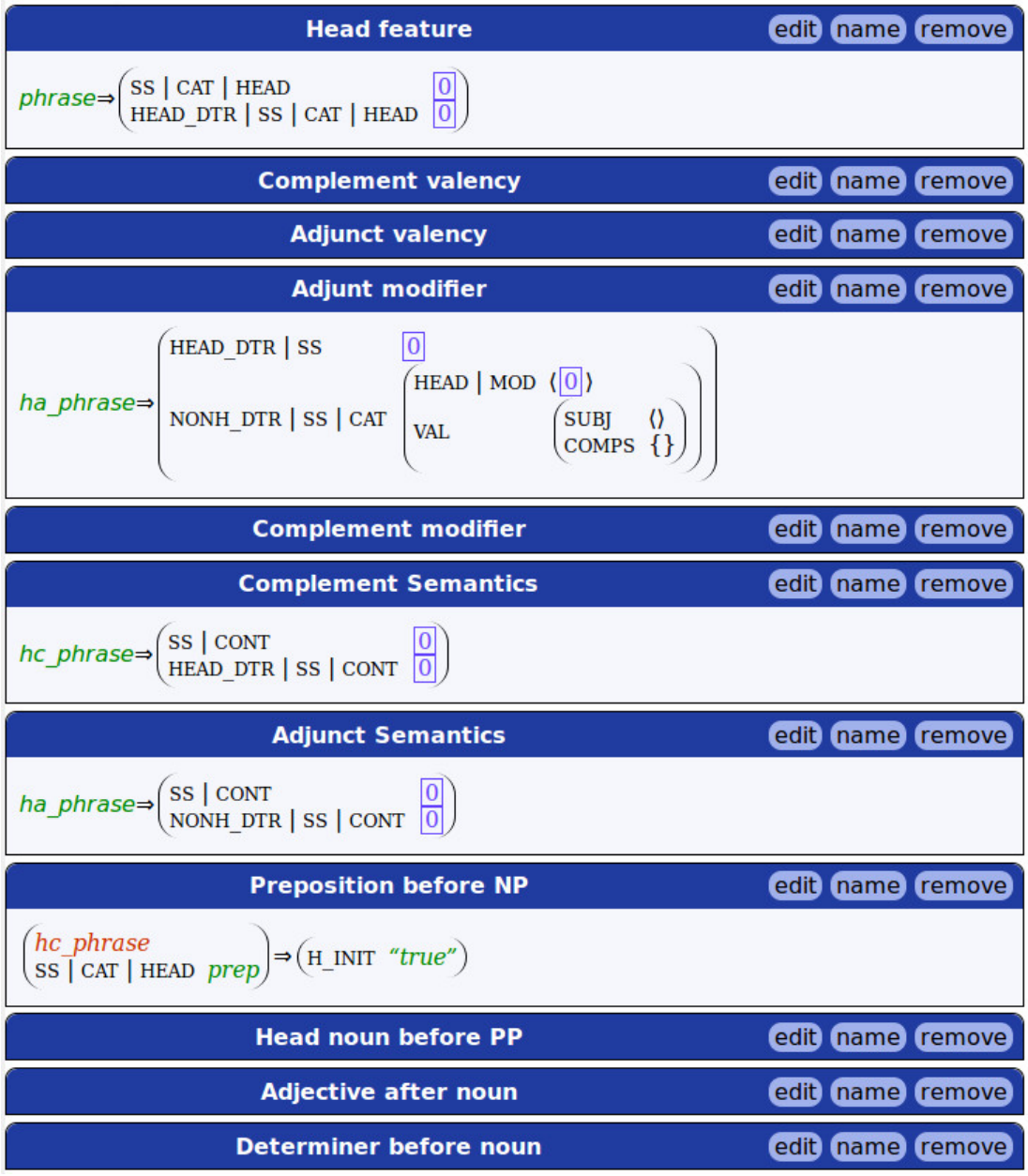


Figure 6.12: Some of the principles

- **Head noun before PP**. In the case of PPs modifying a noun, the prepositional phrase always comes after the noun.
- **Adjective after noun**. In Spanish, the adjective usually comes after the noun, and in this grammar it is always so.
- **Determiner before noun**. Determiners, however, always come before the noun they determine.
- **HC phrase parse order**, **HA phrase parse order**. These principles just eliminate some of the spurious ambiguities that arise from the parsing process, when the parser parsing first the non-head daughter to either the left or right of the head, and then the other way round.

6.2.3 The lexicon

Being a heavy lexicalized formalism, the lexicon is of great importance in an HPSG grammar. Elements in the lexicon are not just types or syntactic categories, but rather full feature structures which contain all the information the word provides, both syntactically and semantically, to the sentence. The rules and principles put together words and phrases precisely according to the values of the features which are specified in the lexicon. In BOLDE, we can organize entries in the lexicon into paradigms, with morphology and templates for the entries.

Common nouns (Figure 6.13) have four inflected forms, for the different combinations of gender and number. The class entry in the table is used for entries which only have some of the forms. The feature structure for nouns is simple, but common nouns have a special characteristic: they require a determiner. The determiner is thus listed in the valency of the noun, and when found it contributes its semantics to those of the noun in the form of a restriction on its value. The inflection of the noun root is specified on the left in the form of regular expressions, in this simple grammar just concatenating the appropriate morpheme to the root ($ means end of the string).

Verbs in Spanish, as in other Romance languages, have a plethora of inflectional forms (called *conjugación* in Spanish). For time reasons, not many have been encoded in BOLDE, but the few there show how it could be done for a larger paradigm. An example template can be found in Figure 6.14. Verbs have straightforward semantic content, a predicate with a few arguments. These arguments are the content objects of the elements found in the valency. In Spanish,

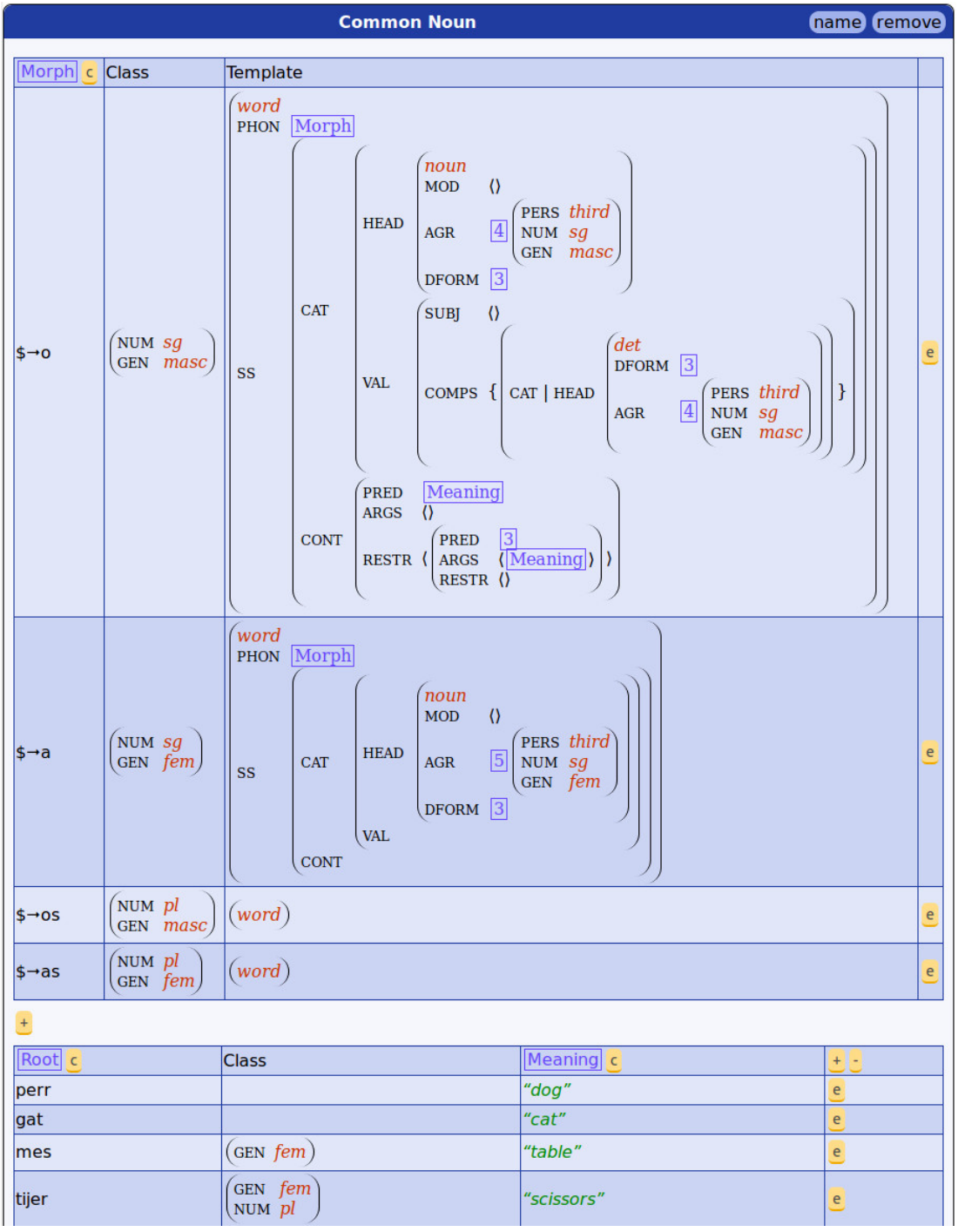


Figure 6.13: The common noun paradigm

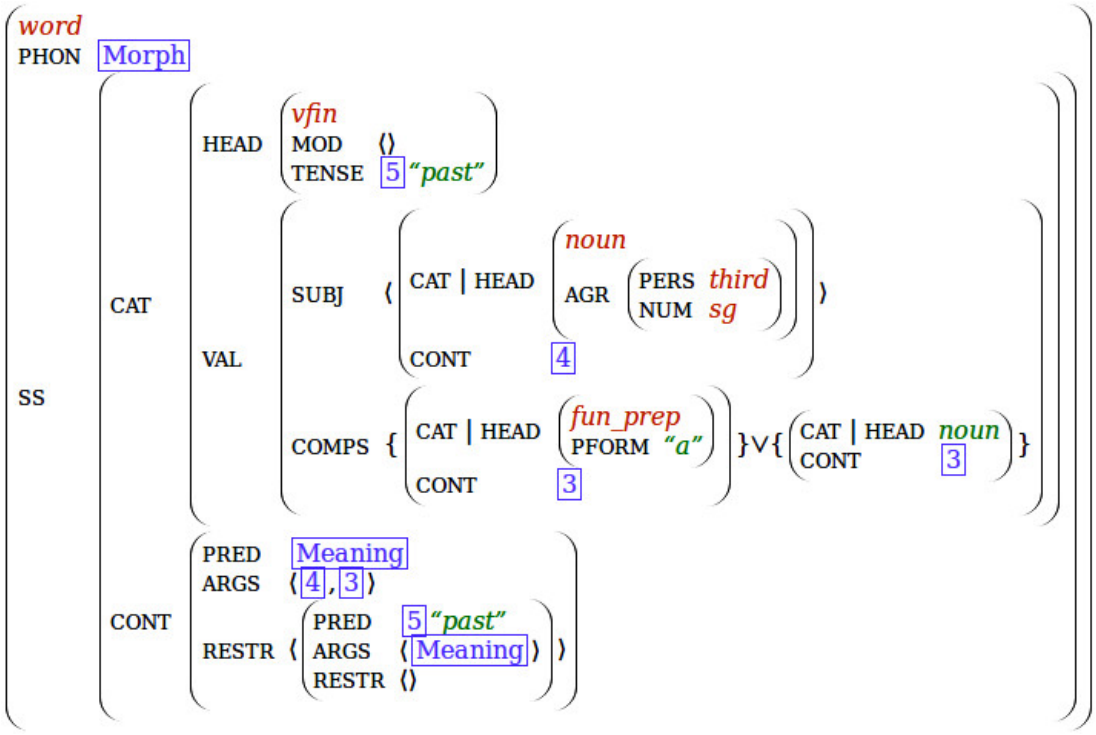


Figure 6.14: The verbal paradigm

the direct object is sometimes found as a prepositional phrase with 'a' (to) as the `pform`, and thus the `comps` feature is a disjunction. Tense in this grammar is encoded as a restriction on the verb predicate.

Adjectives are also flective words. They have the same four combinations as common nouns, with the same morphology, and as can be seen in Figure 6.15, they have a noun in their `mod` feature, with which they agree in gender and number. Their semantic contribution is the same as that of the modified noun, but adding a restriction, that of the meaning of the adjective. An interesting new property of adjectives is that some of them can become adverbs. In English this is made by concatenating *ly*, in Spanish the corresponding morpheme is *mente*. in Figure 6.16 we can see the paradigm entry for this derivation, which also introduces the template for verb-modifying adverbs. The class guard (ADVERB "yes") lets us specify which adjectives should follow this rule and which should not.

Determiners (Figure 6.17) have very simple feature structures, but they are highly irregular, allowing us to demonstrate more of the use of class guards. In the determiner paradigm, the class is used to specify which forms follow the regular inflectional rules. The irregular forms must of course be listed apart, and their agreement feature manually encoded.

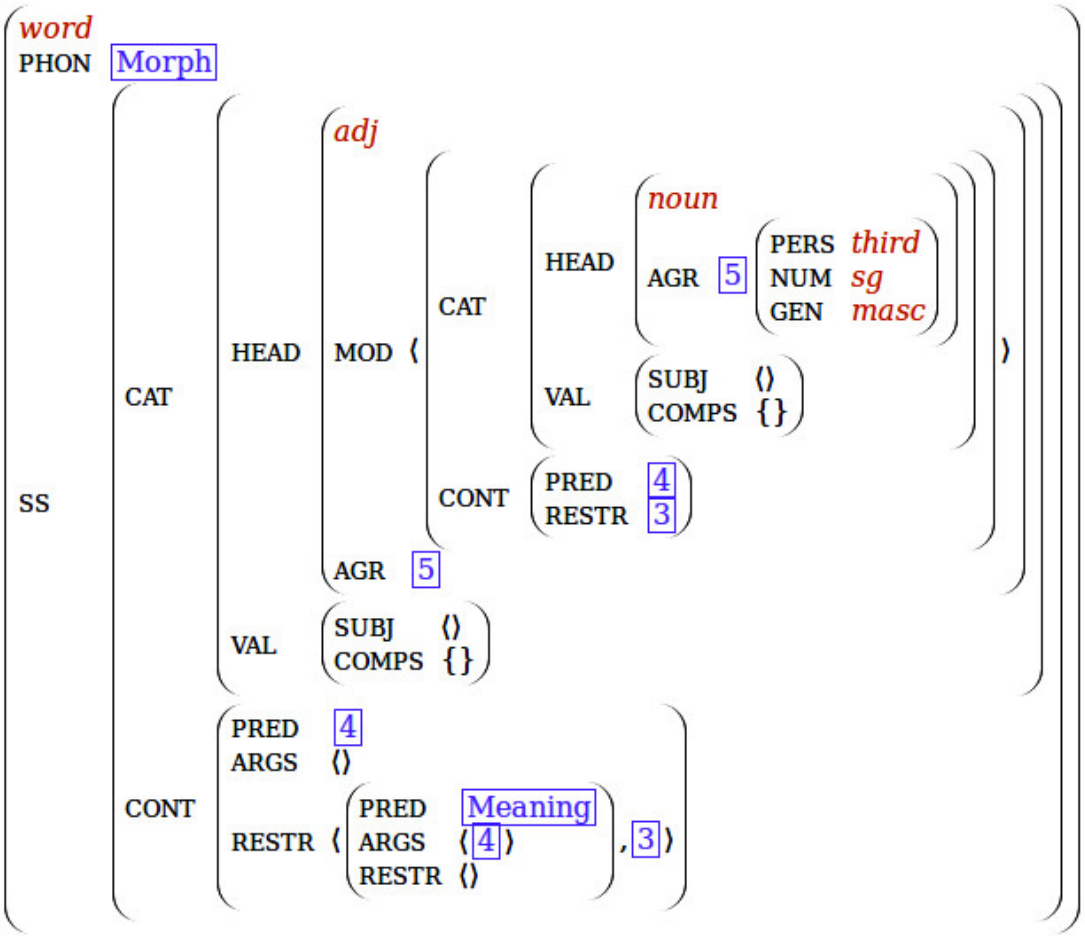


Figure 6.15: The adjective template for masculine singular

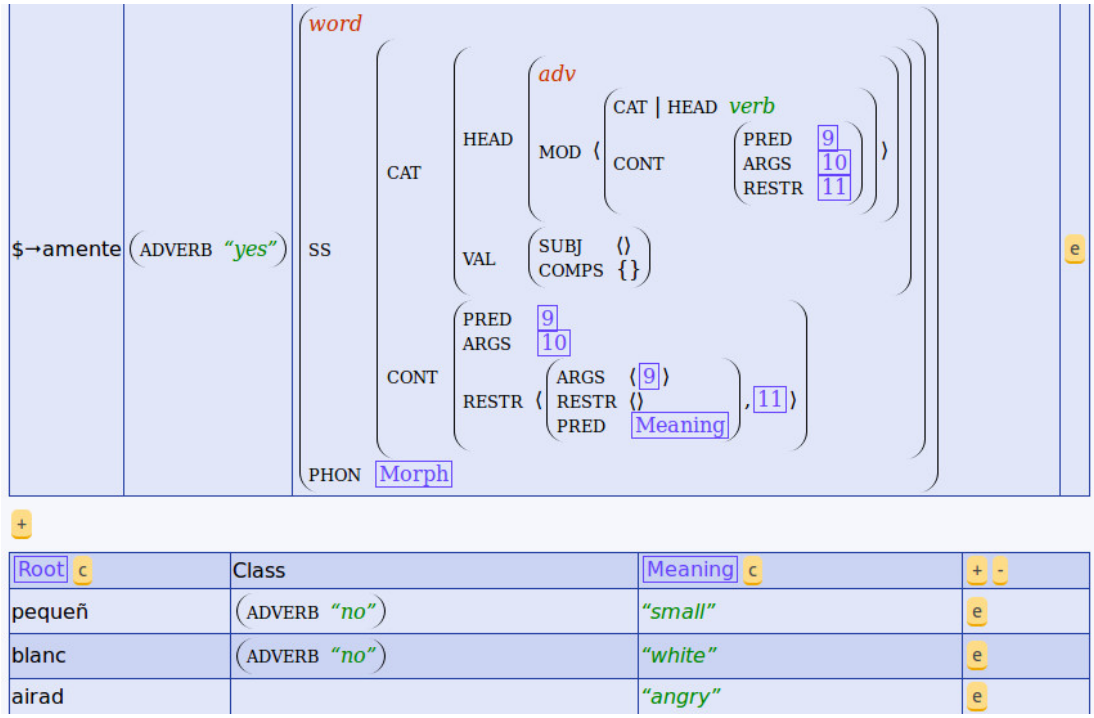


Figure 6.16: Deriving adverbs from adjectives

The next paradigm is that of Proper Nouns, not very interesting in our syntactic implementation. The difference from common nouns is that they do not inflect, and they do not need a determiner in the valency feature.

More interesting are prepositions. They can have non-empty valency and mod features at the same time. In the valency they seek a noun phrase, which makes the rest of the prepositional phrase. There are then two versions: functional prepositions just propagate the semantics of the noun phrase, adding the syntactical information of the `pform`. Lexical prepositions, however, have non-empty `mod` features. They modify another word, for example a verb, adding a restriction to its semantics. The class lets us specify whether a given preposition can act as lexical, functional or both. This paradigm can be seen in Figure 6.18.

Two paradigms follow for adverbs, those modifying a verb and those that modify another modifier (adjective or adverb). The distinction is due to the way the semantics are handled, making modifier modifying adverbs add a restriction to the restriction added by their head. The last paradigm in the grammar is that of prepositional complement verbs, called *Complemento de régimen preposicional* in Spanish. These are verbs whose complements are prepositional phrases headed by a particular preposition. We have already seen this in transitive verbs, since the direct object can be headed by the preposition *a*, so they bring nothing new.

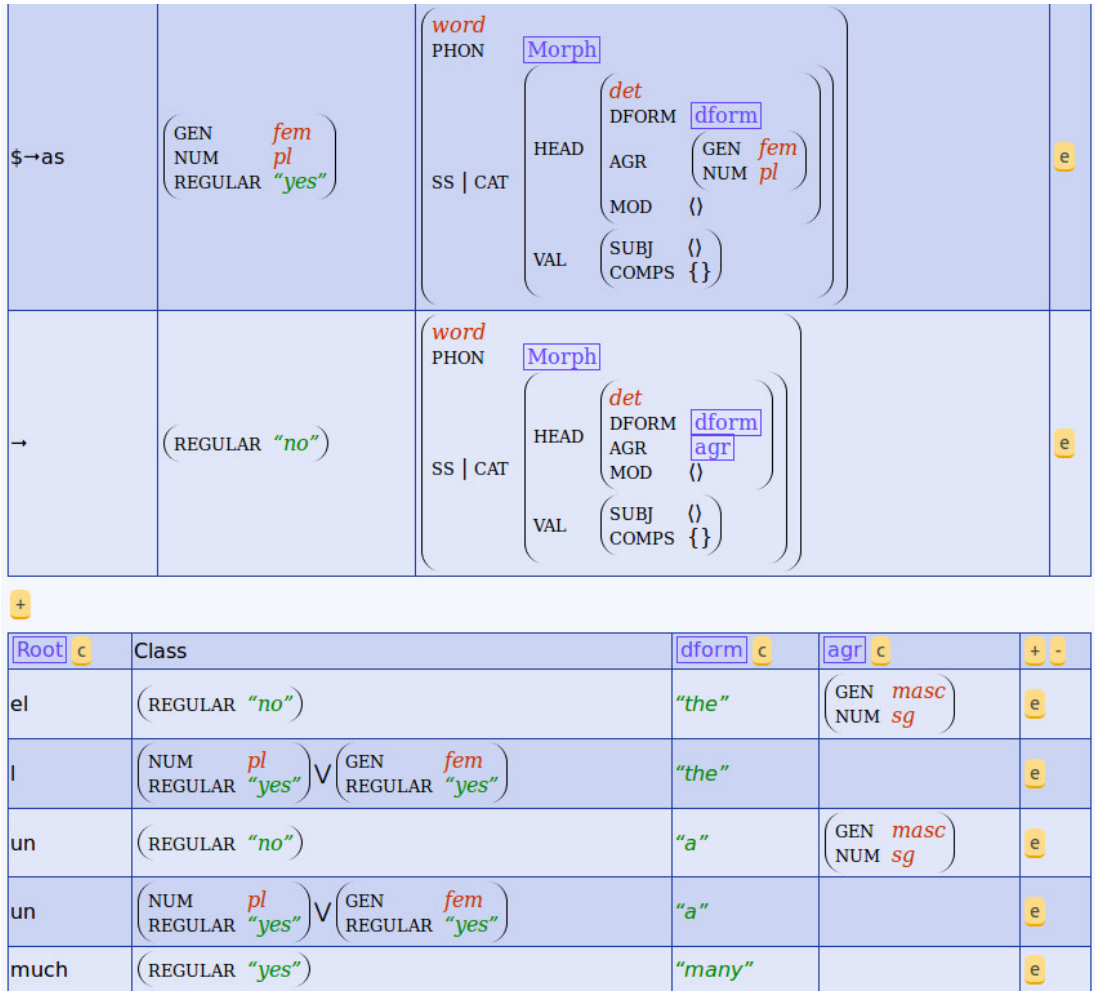


Figure 6.17: Determiner inflection

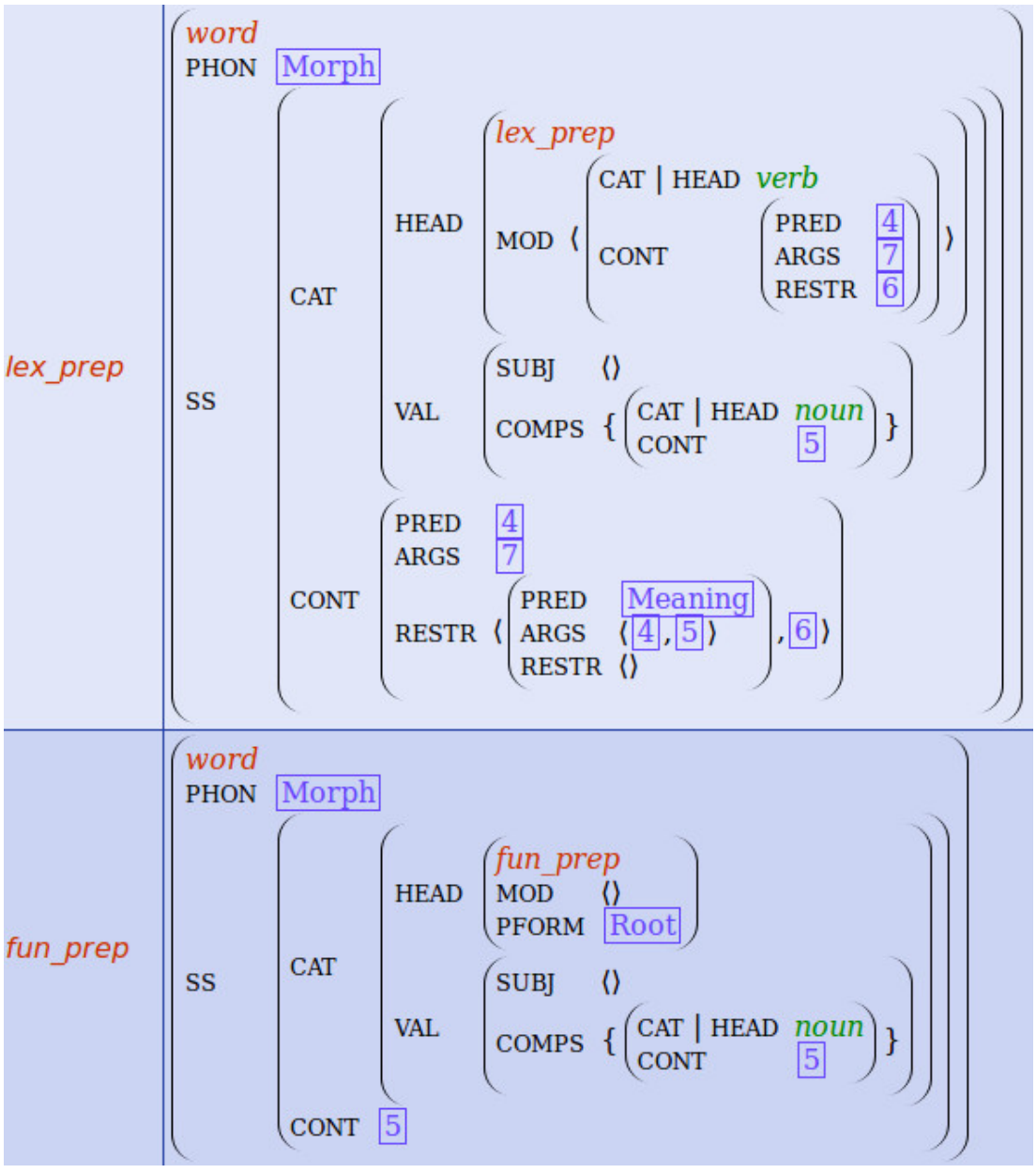


Figure 6.18: Preposition paradigm

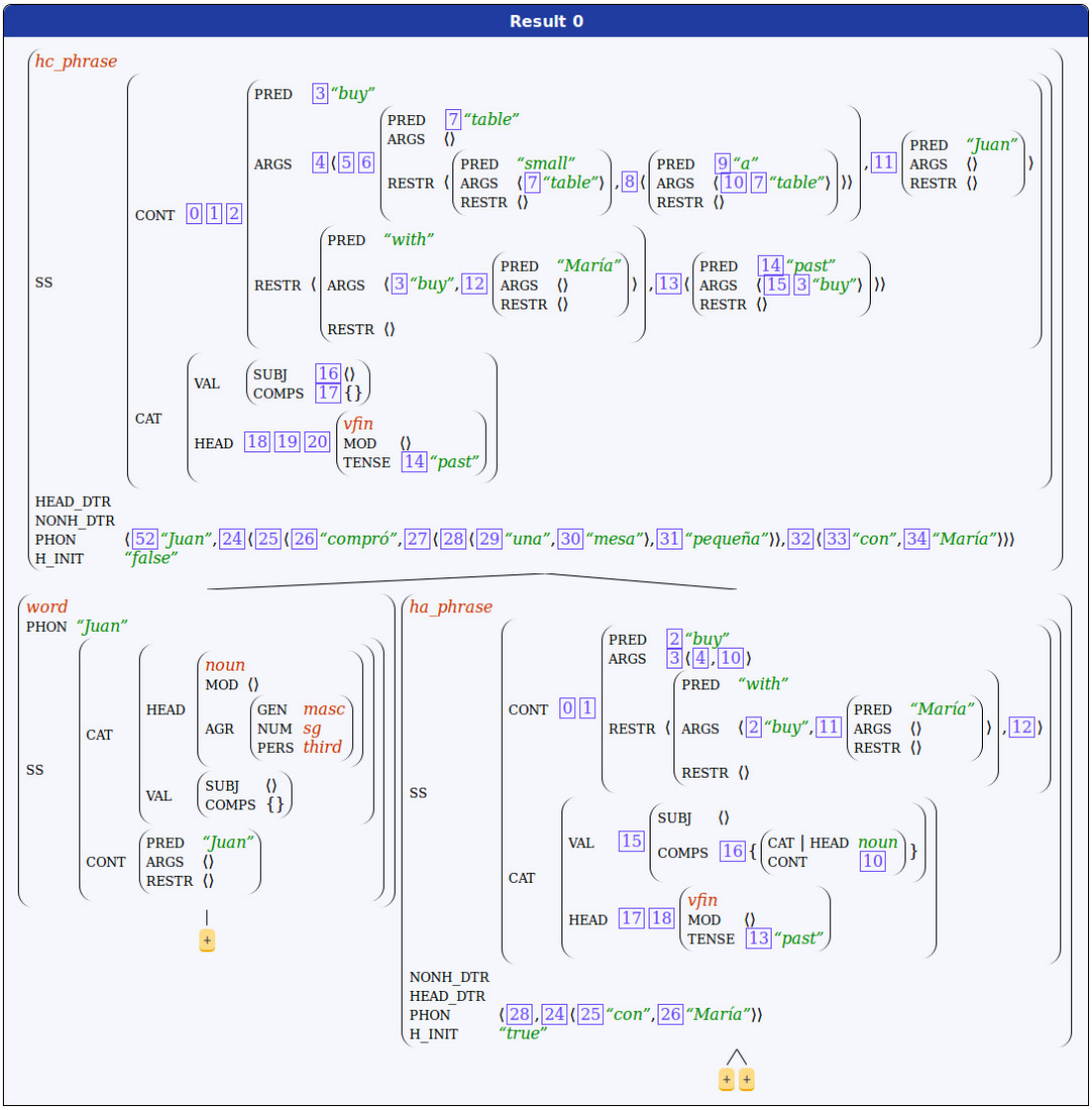


Figure 6.19: The top of the derivation tree for "Juan compró una mesa pequeña con María" (Juan buys a small table with María)

6.2.4 Parse trees

Parsing a series of test sentences with this HPSG grammars gives as results the sentence node, a very big feature structure which itself contains the full derivation history thanks to the `head_dtr` and `nonh_dtr` features. However, Borjes also provides the partial structures that were produced during the parse, as we can see in Figure 6.19. This lets us examine the parsing progress and detect errors or possible improvements. Nonetheless, the root of the tree contains the expected HPSG feature structure for the whole sentence, with all variables bound, the syntactic and semantic features, and the phon list (Figure 6.20).

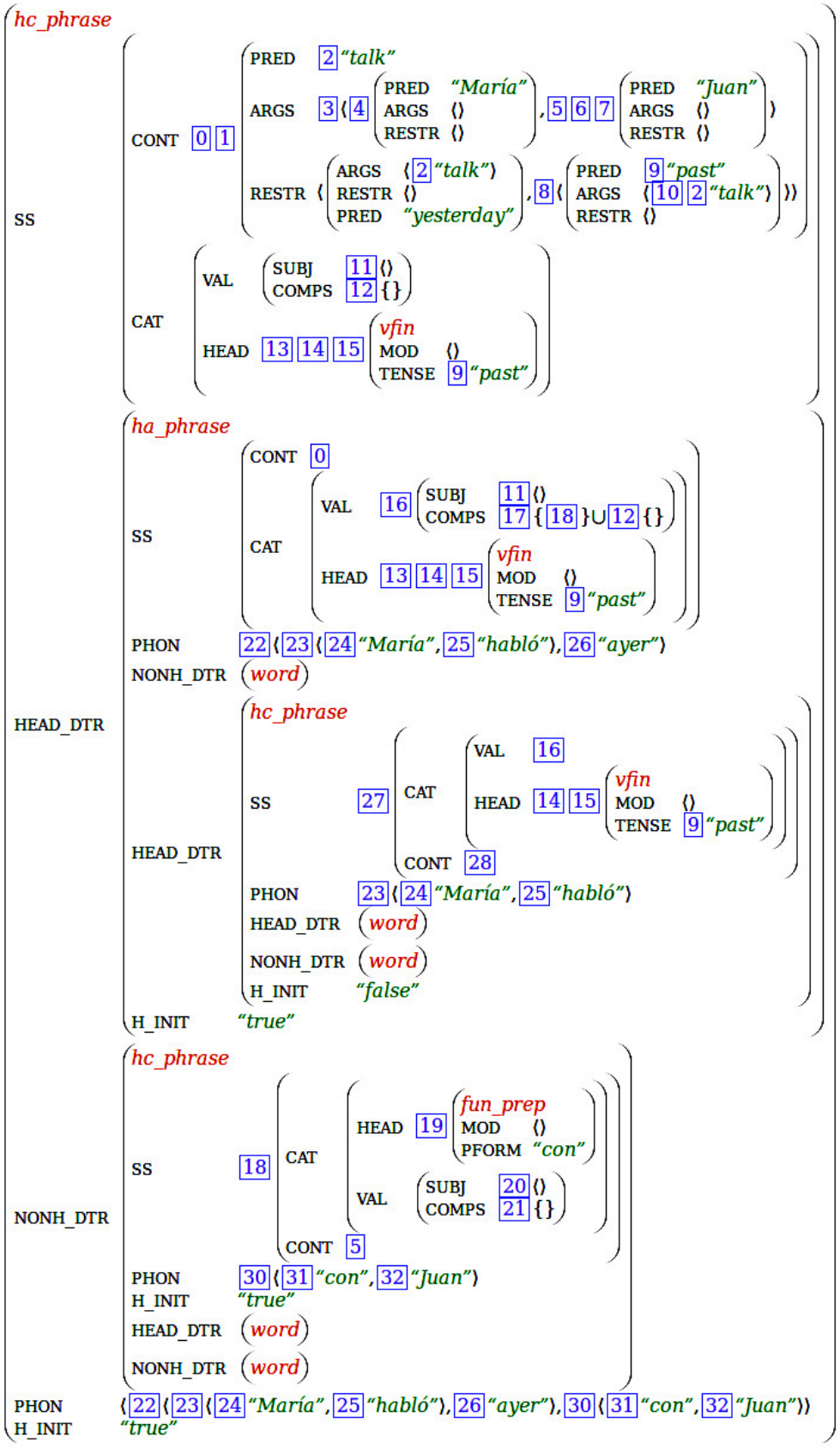


Figure 6.20: The feature structure for "María habló ayer con Juan" (María talked yesterday with Juan)

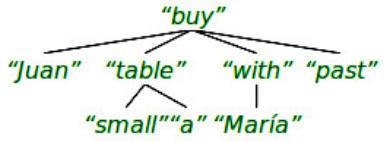


Figure 6.21: Juan compró una mesa pequeña con María

Figure 6.22: Juan observó muchos perros muy airados

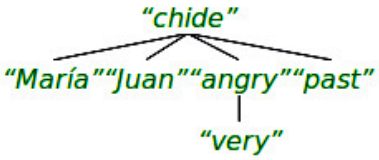


Figure 6.23: María regañó muy airadamente a Juan

Figure 6.24: Juan amó mucho Madrid

Figure 6.25: Semantic trees for the Spanish sentences

6.2.5 An extension: semantic trees

Another thing we can do with the results from our parsing is pass them through a custom filter. As we saw in the previous test case, we can write a javascript file and place it in the pipeline, so that all the objects coming out of the parser are first processed by our code. The code for the javascript engine has access to the Borjes library, so it can extract features from the parse trees, and convert them into something more wieldy.

In the file `semantics.js` that is precisely what has been done. It is a moderately complex piece of code, that can be read in Listing 6.7. The process first extracts the root node of the tree, and then finds its semantic representation, under the `cont` feature of the synsem. Finally, it traverses the predicate and restrictions, and converts them into a deep semantic tree. This tree is then output to be displayed. We have thus managed to parse the Spanish sentences into a semantic tree representation, with the meanings encoded in English, as can be seen in Figure 6.25.

```
var B = Borjes.types;

var FStruct = B.FStruct;
var World = B.World;
var Tree = Borjes.Tree;
var e_list = B.List();

function Cont_to_Tree (cont, world, is_restr) {
    if (cont.borjes !== 'fstruct' && cont.borjes !== 'tfstruct') {
        throw 'unsaturated valency';
    }
    var pred = World.resolve(world, FStruct.get(cont, 'pred'));
    var args = World.resolve(world, FStruct.get(cont, 'args'));
    if (is_restr) {
        args = World.resolve(world, args.rest);
    }
    var children = [];
    while (args.borjes && args.borjes === 'list') {
        var arg = World.resolve(world, args.first);
        children.push(Cont_to_Tree(arg, world));
        args = World.resolve(world, args.rest);
    }
    var restr = World.resolve(world, FStruct.get(cont, 'restr'));
    while (restr.borjes && restr.borjes === 'list') {
        var re = World.resolve(world, restr.first);
        children.push(Cont_to_Tree(re, world, true));
        restr = World.resolve(world, restr.rest);
    }
    if (children.length === 0) {
        return B.copy(pred);
    }
    return Tree(B.copy(pred), children);
}

for (var i = 0; i<input.length; i++) {
    if (input[i].node) {
        var p = input[i].node;
        if (p !== null && p.borjes && p.borjes === 'tfstruct') {
            try {
                var w = p.borjes_bound;
                var ss = World.resolve(w, FStruct.get(p, 'ss'));
                var cont = World.resolve(w, FStruct.get(ss, 'cont'));
                var t = Cont_to_Tree(cont, w);
                World.bind(w, t);
                output(t);
            } catch (e) {}
        }
    }
}
```

Listing 6.7: Semantic filter for the HPSG parses

6.3 Testing

In light of the implementation discussed in chapters 4 and 5, and given the possibilities demonstrated with the previous examples, we can evaluate the work according to the aims and objectives we set in chapter 1.

- **Linguistically effective**. BOLDE, with the aid of Borjes, is capable of doing much linguistic work. It is centered on formal grammar parsing, but it can be extended to support other operations. Some of these extensions could be easily incorporated into the system too.

- **Accessible**. As a web application, BOLDE only requires that the user has a web browser installed, which most (if not all) already have.

- **Intuitive**. The visual editor makes editing complex unification-based grammars easy and intuitive, doable with a mouse and keyboard and without needing any programming abilities. However, this is the only part of the system that could be made graphical, while other parts (like the pipeline editor) could have benefited from a graphical interface too.

- **Collaborative**. Live editing of files, such as that performed by Google Docs or ShareLatex has been achieved. Files are also automatically saved to the cloud, which also fulfills the accessibility goal. The system lacks, however, some social features, which are not too complicated to implement but there was no time for it.

- **Open**. All code is open-source, and uses modern and standard technologies. This should make it as open and broadly accessible to developers as possible. It can also be improved via code plugins, which makes it amenable to modifications and improvements by the users, in order to support the workflow and operations they need.

- **Cross-platform**. All platforms where a modern, standards-compliant browser can be installed are supported. Thanks to the compilation of the client, many modern features are also compiled into versions understandable for older browsers.

  - Windows 7-10: ✓ (Firefox, Google Chrome)
  - Linux: ✓ (Firefox, Google Chrome)
  - Mac OS: ✓ (Firefox, Google Chrome) ✗ (Safari not supported)

  - Android: ✓ (Firefox) ✗ (native Android browser not supported)
  - iOS: ✗ (Not yet tested)

- **Open source**. All software used for the system, and the system itself, are open source, and how to access them is stated in the source code (`package.json` files) and this document.

- **Live editing**. Live editing is achieved thanks to `sharejs`. Edit operations are transmitted in almost realtime, taking basically the roundtrip time of HTTP requests between the two clients.

- **Persistent**. All files are saved in the server. Save for server corruption or other cases of force majeure, changes are persistent. Text files can also be copied, for local system saving. Exporting and importing files easily, especially the visual grammars, has not been implemented.

- **Visual**. As mentioned before, UBG grammars and their lexicon can be edited visually. The parse results are also displayed graphically in the form of derivation trees, and feature structures. Some more components might benefit from graphical interaction, such as the pipeline editor.

- **Powerful**. While difficult to measure, the objective that was set was to be able to offer all the power of the UBG grammatical formalism. Borjes supports all of the characteristics of this formalism, apart from bugs and error that might yet be discovered.

- **Empirical**. Mathematical and algorithmical constructs are first class citizens in Borjes thanks to predicates, as has been seen in section 6.1. Predicates remain to be implemented in the visual editor, however.

- **Configurable execution**. The pipeline function is generic, and any components can be connected in any configuration. However, only one pipeline is supported, and it might help to have at the same time more than one possible execution path.

- **Extensible**. The javascript engine in BOLDE allows the user to plug into the system any extension code they might need. Support for connecting to third-party resources, however, has not been tested, and is complicated enough that the system should offer shortcuts for doing it.

- **Interoperable**. Due to time constraints, this objective has not been realized at all, and BOLDE resources are not compatible with the formats used by other platforms.

- **Multilingual**. Borjes works with Unicode characters, so any language which can be encoded in Unicode can be used in BOLDE. The HPSG grammatical formalism is also general enough to accommodate languages as different as English and Japanese, and in theory such grammars could also be written in BOLDE. This has not been tested, but a simple grammar for Spanish has been demonstrated in section 6.2, which uses characters outside the ASCII range.

Chapter 7

Conclusions and future work

7.1 Conclusions

The first conclusion we can draw from the work done is that the project was quite ambitious. Building a web environment is quite a task in itself, and on top of it, much scaffolding and deep foundations had to be built. In the end, the software has been developed, and it covers all the areas that were intended from the beginning. However, even if broad, coverage is sometimes shallow.

There are many areas which have not been finished in a satisfactory manner. The graphical aspect, for example, does not cover pipeline editing, and still misses exposing some of the functionality of the linguistic engine underneath in the visual editor.

The collaborative platform has all the connections and components such software might need, but it feels empty. There is no social facet, and user interaction with the system is limited to the development of the linguistic entities, there not being any project management capabilities.

It is also true that the collaborative aspect has not been developed to its full potential. It is difficult to test a team-oriented platform with only one user working on it, and it is even more difficult to get users to work on an unfinished environment, still in development. Obvious features such as live editing have been completed, but with more active users, other not-so-obvious improvements might be found and implemented.

On the other hand, mainly due to the limited time available, the products that were intended to be developed on the platform have not been as completely realized as was expected. Time has been spent, mostly, on the difficult task of developing both the linguistic engine, Borjes, and the development environment, BOLDE.

As such, statistical methods or algorithms have not been written or tested in the framework. The power to build them is there, as was demonstrated in the previous chapter, but developing them would require a time that is not available. The HPSG grammar of Spanish is very small. It analyzes many basic phenomena, but the true power of sophisticated grammar formalisms is in modeling advanced linguistic issues, and there was no time for doing that.

Despite all the problems mentioned, the net outcome for the work is positive. The platform, even if small yet, can be considered a success in the terms of a software application. It supports a wide variety of user environments, it is easily accessible, and is reasonably stable.

When testing it, and developing the different examples, BOLDE's automatic saving, instant access on any device with an Internet connection, and accessible development model were deemed to be actually useful. The workflow and development paradigm they enable are positive features, which actually help with getting the work done.

The extensibility of the platform, in the form of code extensions, and its modular architecture, feel also very comfortable to the modern developer. They follow the paradigms and customs of current web development, and writing the semantic translation for the second use case was straightforward and surprisingly easy.

In part, this is thanks to the Borjes library, which is a very generic formal grammar framework. It has a clear interface, and for any developer used to Javascript programming, it is easy to get started with. It gives clear access to the internals of the parser and its linguistic representations, and its objects are native Javascript values. Of course, the same might be said of the platforms implemented in Common Lisp or Prolog for the accustomed programmer. However, Javascript is a newcomer in this area, and many of its users are not comfortable in such a different paradigm as logic programming. Thus, Borjes might help make formal computational linguistics more accessible to a whole new sector of computer scientists.

Another positive return of using Javascript is its tight integration with graphical interfaces, thanks to the browser. While the visual editor might seem somewhat incomplete, it does already offer a lot of functionality. It remains to be seen if graphical editing would really be an aid to the development of large grammars, but in the author's opinion it makes for a great introduction to the formalism, without the need to learn a programming language. Moreover, all other platforms offer visualization of results, since these are often very complex, so the visual ca-

pabilities of the platform are actually a necessity. Having them in the form of web pages opens up many new possibilities of storing, displaying and interacting with linguistic data.

Going back to Borjes, it should be remarked that its generic character would allow for implementing other formalisms, such as LTAG or LFG, by merely translating their rules and constraints into the Borjes dialect. The type system is also very comfortable to use for doing numerical and big data analysis, within the grammar itself, without requiring pre- or post-processors.

If Borjes were to be rewritten, a few design decisions would be reconsidered, though. Not enforcing well-typedness in the grammar does not seem to make the parsing process lose any properties, but it misses opportunities on helping the development process. By not telling the engine what the meaning of types is in the signature, it cannot do helpful things like auto-suggest possible values during an edit, detect misspellings or logical inconsistencies, or auto-fill content which is of no particular interest, but must be there anyway for the formalism to work.

The unification mechanism might need to be reconsidered, too. In modern functional programming, there is a trend of trying to make more and more of the state of an application immutable. This might help prevent the necessity of copying Borjes objects continuously, to defend against unintended modifications. The type system is also a never-ending fight with Javascript itself, which as a scripting, interpreted language, offers very little help for the developer to stay in control of the data.

Nonetheless, the work on Borjes has been enormously rewarding, and its simple and clear architecture allow for it to work with very complicated constructs such as lattices, free and bound variables, and disjunctions. Even with bugs probably waiting to be discovered, and with some polishing and finishing up still to be done, it is a flexible and powerful library. A library for doing formal computational linguistics, in a modern language such as Javascript.

7.2 Future work

Some of the possibilities for continuing the work of this master thesis have already been outlined above. The principal one is to get the system in use, so that more users can find its pros and cons, its strengths and weaknesses, and drive development forward.

Apart from that general aim, more detailed and immediate problems await. One of the powers of an interpreted language, such as Javascript, and a dynamic

parser such as that of Borjes, is that debugging can be easily done during execution. This was a driving goal of Borjes, and there is support for it in the library in the form of hooks and callbacks for plugging into the internal work of the parser. The potential is not exploited by BOLDE, and it probably requires some work to get integrated.

One of the solutions mentioned in the design chapter, in section 3.3, was to make the engines work both in the client and the server. That way, quick operations could be done locally, while lengthier and more expensive ones could be sent to the server. This would also help more limited platforms such as mobile devices (even though it has been found that they are powerful enough to support the parsing process). This dual solution for engines has not been implemented, but implementing it should not be too difficult.

On the client side, many small and some big improvements remain to be done. The pipeline should be graphically editable. Some more consistency in the user interface could be achieved, and the aforementioned social features integrated. The treebank module is very incomplete, and it could see much improvement.

With regards to Borjes, work can be always done to improve it, and there are many features that can be integrated into the library. However, the main challenge ahead of it is seeing actual large scale use, so that errors can be found and fixed, and inconsistencies and design decisions ironed out.

Regarding the use of the system, and to further demonstrate its usefulness, developing a statistical application in BOLDE would be very interesting. It would truly show the advantages of having such a platform, especially if connecting to third-party resources over the internet, or making use of large corpora, is achieved.

It would also be interesting to further develop the grammar of Spanish. NLP is a growing trend in computer science, seeing more and more use in consumer applications and industrial systems. Knowledge-based computational linguistics has a hard time reaching the general public, though. A javascript, straightforward grammar, of a widely used language such as Spanish, could actually be of use to engineers around the world, and narrow the gap between industry and academia in this more theoretically-minded paradigm.

7.2.1 Evaluation

A final remark would have to be made in regards to evaluation. As has been mentioned, evaluating a system centered on enabling its users is not trivial, and requires resources not easily found within the scope of a software development master thesis.

A big group of users would be needed, with the requirements that BOLDE aims to solve: a multi-person project on computational linguistics, where a technically advanced platform would help but synchronization and collaboration were also important. Such a group might be found in academic research on computational linguistics, or perhaps in the teaching of it.

Once these users were found, evaluation could proceed by measuring the success of the group in pursuing their project. Their comments and suggestions in respect to the usability of the platform would also be a valuable aid to its development, and a useful measure of the suitability of the platform to the objectives that were originally proposed.

Bibliography

Ahn, Wonsun et al. (2014). "Improving JavaScript performance by deconstructing the type system". In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM, pp. 496–507.

Alexander, Tormasov and Alexey Kuznetcov (2002). *Use sendfile() to optimize data transfer — TechRepublic*.

Anthes, Gary (2012). "HTML5 leads a web revolution". In: *Communications of the ACM* 55.7, pp. 16–17.

Armbrust, Michael et al. ((2010)). "A view of cloud computing". In: *Communications of the ACM* 53.4, pp. 50–58.

Ben-Kiki, Oren, Clark Evans, and Brian Ingerson (2009). "YAML Ain't Markup Language (YAML™) Version 1.1". In: *Working Draft 2008-05* 11.

Bird, Steven, Ewan Klein, and Edward Loper (2009). *Natural language processing with Python.* " O'Reilly Media, Inc.".

Bos, Bert et al. (2005). "Cascading Style Sheets, level 2 revision 1 CSS 2.1 Specification". In: *W3C working draft, W3C, June*.

Carpenter, Bob (2005). *The logic of typed feature structures: with applications to unification grammars, logic programs and constraint resolution*. Vol. 32. Cambridge University Press.

Chomsky, Noam (1959). "On certain formal properties of grammars". In: *Information and control* 2.2, pp. 137–167.

CommonJS module specification (2015). URL: `http://www.commonjs.org.mirrors.page.ca/` (visited on 09/23/2015).

Copestake, Ann (2002). *Implementing typed feature structure grammars*. Vol. 110. CSLI publications Stanford.

Copestake, Ann et al. (2005). "Minimal recursion semantics: An introduction". In: *Research on Language and Computation* 3.2-3, pp. 281–332.

Crockford, Douglas (2006). "The application/json media type for javascript object notation (json)". In:

Ecma, ECMA (1999). "262: Ecmascript language specification". In: *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*

ECMAScript, ECMA, European Computer Manufacturers Association, et al. (2011). *ECMAScript Language Specification.*

Google Docs (2015). URL: `https://docs.google.com/` (visited on 01/18/2015).

Gosling, James (2000). *The Java language specification.* Addison-Wesley Professional.

Higginbotham, James (1987). "English is not a context-free language". In: *The Formal Complexity of Natural Language.* Springer, pp. 335–348.

Järvinen, Hannu (2011). "HTML5 Web Workers". In: *T-111.5502 Seminar on Media Technology BP, Final Report*, p. 27.

Kambona, Kennedy, Elisa Gonzalez Boix, and Wolfgang De Meuter (2013). "An evaluation of reactive programming and promises for structuring collaborative web applications". In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications.* ACM, p. 3.

MacFarlane, John et al (2014). *CommonMark.* URL: `http://commonmark.org/` (visited on 09/23/2015).

McCarthy, John (1960). "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Communications of the ACM* 3.4, pp. 184–195.

NLTK – Natural Language Toolkit (2015). URL: `http://www.nltk.org` (visited on 09/30/2015).

NodeJS (2015). URL: `https://nodejs.org/en/` (visited on 09/21/2015).

NPM - the node package manager (2015). URL: `https://www.npmjs.com/` (visited on 09/13/2015).

Oepen, Stephan (2002). *Collaborative Language Engineering: A Case Study in Efficient Grammar-based Processing.* CSLI Publications.

Penn, Gerald and Mohammad Haji-Abdolhosseini (2003). "The Attribute Logic Engine: User's Guide–with TRALE extensions". In: *Revision of the manual: Version* 4.

Pichai, Sundar and Linus Upson (2009). "Introducing the google chrome os". In: *The Official Google Blog* 7.

Pollard, Carl and Ivan Sag (1994). *Head-driven phrase structure grammar.* University of Chicago Press.

Pullum, Geoffrey K and Gerald Gazdar (1982). "Natural languages and context-free languages". In: *Linguistics and Philosophy* 4.4, pp. 471–504.

Sag, Ivan A (2003). "Coordination and underspecification". In: *HPSG (2003a)*, pp. 267–291.

ShareLaTeX (2015). URL: `https://www.sharelatex.com` (visited on 01/18/2015).

Shieber, Stuart M (1987). *Evidence against the context-freeness of natural language*. Springer.

— (1992). *Constraint-based grammar formalisms: parsing and type inference for natural and computer languages*. MIT Press.

Tsujii, Jun'ichi (2011). "Computational linguistics and natural language processing". In: *Computational Linguistics and Intelligent Text Processing*. Springer, pp. 52–67.

List of Figures

3.1 Global architecture 17
3.2 Flux architecture diagram 19
3.3 Example of regions with results 20
3.4 Example of simultaneously editing two files 21

4.1 BOLDE server data flow 27
4.2 Screenshot of the visual editor 36
4.3 Screenshot of the paradigm editor 37

5.1 Borjes architecture overview 42
5.2 The lattice created in Listing 5.3 46
5.3 A big feature structure in borjes-react 67
5.4 Editing in borjes-react 68

6.1 Screenshot of the parse results 73
6.2 A parse tree for a long sentence 73
6.3 A parse tree of LCFG 75
6.4 A PCFG tree 76
6.5 A diagram of the pipeline in Listing 6.6 79
6.6 Treebank searching results for VP(VP(VP(VP))) 80
6.7 Treebank searching results for VP(_,VP) 80
6.8 The full signature for the grammar 82
6.9 Hierarchy of signs 82
6.10 The hierarchy of head types 83
6.11 The four versions of the rules 84
6.12 Some of the principles 85
6.13 The common noun paradigm 87
6.14 The verbal paradigm 88
6.15 The adjective template for masculine singular 89
6.16 Deriving adverbs from adjectives 90
6.17 Determiner inflection 91

6.18 Preposition paradigm 92
6.19 The top of the derivation tree for “Juan compró una mesa pequeña con María” (Juan buys a small table with María) 93
6.20 The feature structure for “María habló ayer con Juan” (María talked yesterday with Juan) 94
6.21 Juan compró una mesa pequeña con María 95
6.22 Juan observó muchos perros muy airados 95
6.23 María regañó muy airadamente a Juan 95
6.24 Juan amó mucho Madrid 95
6.25 Semantic trees for the Spanish sentences 95

Listings

4.1 Default BOLDE configuration 31
4.2 Example of a run.yml file 39
5.1 Example of the use of Borjes modules 42
5.2 Example of the use of primitive types 45
5.3 Example of the use of lattices 47
5.4 Example of the use of feature structures 49
5.5 Example of the use of typed feature structures 50
5.6 Example of the use of lists 51
5.7 Example of the use of worlds and variables 53
5.8 Example of the use of disjuncts 55
5.9 Example of the use of sets 56
5.10 Example of the use of rules 57
5.11 Example of the use of principles 58
5.12 Example of the use of the lexicon 60
5.13 Example of the instantiation of a grammar object 61
5.14 Example of the use of the parser 62
5.15 Example of the use of trees 63
5.16 Example of the use of the yaml loader 65
5.17 Example of the use of Borjes-react 68
6.1 CFG grammar for English 71
6.2 Test sentences for the CFG grammars 72
6.3 Simple run.yml file 74
6.4 Lexicalized CFG grammar 75
6.5 Numerical CFG grammar with predicates 77
6.6 A complex pipeline 78
6.7 Semantic filter for the HPSG parses 96