Table of Contents

FreeLing User Manual

This book contains the user manual for FreeLing. This repository contains the source code of the book that can be read at GitBook.

FreeLing is a library oriented to developers, so a large part of the book describes which are the available C++ modules and classes. Developers can find more information in the Technical Reference Manual.

Non-developer users can also find useful information, such as:

- Installation instructions
- Available modules and functionalities
- Format descriptions for configuration and data files
- Information about how to use FreeLing `analyzer` front-end, and available options

Basic Concepts

FreeLing is a C++ library providing language analysis services (such as morphological analysis, date recognition, PoS tagging, parsing, etc.)

The current version provides language identification, tokenizing, sentence splitting, morphological analysis, NE detection and classification, recognition of dates/numbers/physical magnitudes/currency/ratios, phonetic encoding, PoS tagging, shallow parsing, dependency parsing, WN-based sense annotation, Word Sense Disambiguation, Semantic Role Labelling, and coreference resolution. Future versions are expected to improve performance in existing functionalities, as well as incorporate new features.

FreeLing is designed to be used as an external library from any application requiring this kind of services. If the calling application is written in C++ native calls to the library can be performed. Alternatively, APIs are provided to call main FreeLing functionalities from programs in Java, Perl, or Python.

Additionally, a command-line main program with many customization options is also provided as a basic interface to the library, enabling the user to analyze text files with no need of coding any programs to call the library.

What is FreeLing

FreeLing is a developer-oriented library providing language analysis services. If you want to develop, say, a machine translation system, and you need some kind of linguistic processing of the source text, your MT application can call FreeLing modules to do the required analysis.

In the directory `src/main/simple_examples` in FreeLing tarball, some sample programs are provided to illustrate how an application program can call the library.

In the directory `src/main/sample_analyzer` a couple of more complex programs are provided, which can be used either as a command line interface to the library to process texts, or as examples of how to build customized applications using FreeLing.

1 ## What is NOT FreeLing {#what-is-not-freeling}

FreeLing is not a user-oriented text analysis tool. That is, it is not designed to be user friendly, to have a fancy GUI, or to output results with a cute image, or in a certain format.

FreeLing results are linguistic analysis in a data structure. Each end-user application (e.g. anything from a simple syntactic-tree drawing plugin to a complete machine translation system) can access those data and process them as needed.

Nevertheless, FreeLing package provides a quite complete application program `analyzer` that enables an end user with no programming skills to obtain the analysis of a text. See chapter about `analyzer` for details.

This program offers a set of options that cover most of FreeLing capabilities. Nevertheless, much more advantadge can be taken of FreeLing, and more information can be accessed if you call FreeLing from your own application program.

FreeLing library also includes some classes that are able to dump FreeLing data tructures into some common formats, such as XML or CoNLL-like column format. These classes are limited and may not be able to generate any desired output format as they are out-of-the box, but can be adjusted to your needs or used as a starting poit to write your own dumper to your preferred output format.

Supported Languages

The current version supports the following languages(to different extents and accuracy levels, see table below): Asturian (as), Catalan (ca), German (de), English (en), French (fr), Galician (gl), Croatian (hr), Italian (it), Norwegian (nb), Portuguese (pt), Russian (ru), Slovene (sl), Spanish (es), and Welsh (cy).

Table 1.1: Analysis services available for each language.

	as	ca	cy	de	en	es	fr	gl	hr	it	nb	pt
Tokenization	X	X	X	X	X	X	X	X	X	X	X	X
Sentence splitting	X	X	X	X	X	X	X	X	X	X	X	X
Number detection		X		X	X	X	X	X		X		X
Date detection		X		X	X	X	X	X				X
Morphological dictionary	X	X	X	X	X	X	X	X		X	X	X
Affix rules	X	X	X	X	X	X	X	X		X	X	X
Multiword detection	X	X	X		X	X	X	X		X		X
Basic named entity detection	X	X	X		X	X	X	X		X		X
B-I-O named entity detection		X			X	X		X				X
Named Entity Classification		X			X	X						X
Quantity detection		X			X	X		X				X
PoS tagging	X	X	X	X	X	X	X	X		X	X	X
Phonetic encoding					X	X						
WN sense annotation		X			X	X	X	X	X			
UKB sense disambiguation		X			X	X	X	X	X			
Shallow parsing	X	X			X	X		X				X
Full/dependency parsing	X	X			X	X		X	X			
Semantic Role Labelling		X		X	X	X						
Coreference resolution					X	X						

FreeLing is designed to be modular and to keep linguistic data separated from code. So, most modules can be adapted to a new language just replacing a configuration file or providing a file with rules specific for that language.

Thus, most of the missing crosses in previous table could be filled up just writting a small configuration or rule file (this is the case of, e.g. affixation rules, multiword detection, basic named entity detection, quantity detection, phonetic encoding) or an appropriate dictionary/lexicon file (e.g. Morphological dictionary or WN sense annotation modules).

FreeLing also includes WordNet-based sense dictionaries for some of the covered languages, as well as some knowledge extracted from WordNet, such as semantic file codes, or hypernymy relationships. See http://wordnet.princeton.edu and http://www.illc.uva.nl/EuroWordNet for details on WordNet and EuroWordNet, respectively.

See the Linguistic Data section on FreeLing webpage to find out more about the size and origin the linguistic resources for these languages.

See file `COPYING` in the distribution packages to find out the license of each third-party linguistic resource included in FreeLing packages.

License

FreeLing code is licensed under Affero GNU General Public License (AGPL).

The linguistic data collections are distributed under diverse licenses, depending on their original sources.

Find the details in the `COPYING` file in the tarball, or in the License section in FreeLing webpage.

Contributions

FreeLing is developed and maintained by people in TALP Research Center at Universitat Politecnica de Catalunya.

Many people further contributed to by reporting problems, suggesting various improvements, submitting actual code or extending linguistic databases. A detailed list can be found in Contributions section at FreeLing webpage.

Getting it to work

Requirements

To install FreeLing you'll need:

- A typical Linux box with usual development tools:
  - bash
  - make
  - C++ compiler with STL and C++11 support (e.g. g++ 4.6 or newer)
- Enough hard disk space -about 3 Gb for source and temporary compilation files (which you can delete later if needed), plus some 1.1Gb for final installation.
- Some external libraries are required to compile FreeLing:
  - `libboost` & `libicu` libraries. Included in all Linux distributions. You probably do not have all neeeded components installed. Make sure to install both runtime and development packages for:
    - libicu
    - libboost-regex
    - libboost-system
    - libboost-thread
    - libboost-program-options
    - libboost-locale (only required for MacOSX or FreeBSD, not required in Linux)
  - `libz` compression library. Included in all Linux distributions. You probably do not have all neeeded components installed. Make sure to install both runtime and development packages for:
    - zlib

Orientative package names

The name of the packages containing the dependencies listed above vary depending on your linux distribution.

Please check the package manager in your system, and use its package search capabilities to install the needed dependencies.

As an orientation, here are the names of these packages in some popular distributions. (Note that this may change over time too)

- Ubuntu/Debian: `libboost-dev libboost-regex-dev libicu-dev libboost-system-dev`

`libboost-program-options-dev libboost-thread-dev zlib1g-dev`

- OpenSuse/Fedora/Mandriva: `boost-devel boost-regex-devel libicu-devel boost-system-devel boost-program-options-devel boost-thread-dev zlib-devel`
- Slackware: `boost icu4c zlib`

Note that you need to install both the binary libraries and the development packages (usually sufixed as `-dev` or `-devel`). Most package managers will install both binary and development packages when the `-dev` package is required. If this is not your case, you'll need to manually select both packages.

See details on the installation procedure in section Installation.

Installation

This section provides a detailed guide on different options to install FreeLing (and all its required packages).

Install from .deb binary packages

This installation procedure is the fastest and easiest. If you do not plan to modify the code, this is the option you should take.

Binary packages are available only for stable FreeLing versions. If you want to install an alpha or beta version, please see section about installing from GitHub.

The provided packages will only work on debian-based distributions. They have been tested in Ubuntu (12.04-LTS Precise, 14.04-LTS Trusty, 16.04-LTS Xenial) and Debian (7.0 Wheezy, 8.3 Jessy, 9.0 Strecht).

Most debian-based systems will launch the apropriate installer if you just double click on the package file. The installer should solve the dependencies and install all required packages.

If that doesn't work, you can install it by hand (in Ubuntu or Debian) with the following procedure (will probably work for other debian-based distros):

1. Install required system libraries.

   The following commands should install *both* header packages and binary libraries. If they don't, use your package manager to install all required packages as described in section Requirements.

   ```
   sudo apt-get install libboost-regex-dev libicu-dev zlib1g-dev
   sudo apt-get install libboost-system-dev libboost-program-options-dev libboost-thread-dev
   ```

2. Install freeling package
   `sudo dpkg -i freeling-4.0.deb`

   In a Debian system, the above commands must be issued as root and without `sudo`.

Install from `.tar.gz` source packages

Installation from source follows standard GNU autoconfigure installation procedures (that is, the usual `./configure && make && make install` stuff).

Installing from source is slower and harder, but it will work in any Linux box, even if you have library versions different than those required by the `.deb` package.

1. Install development tools

   You'll need to install the C++ compiler and other developer tools:
   `sudo apt-get install build-essential automake autoconf libtool`

   In Debian, use the same command as root, without `sudo`. In other distributions, check the distribution package manager to install a working C++ compiler and autotools.

2. Install packaged requirements

   All required libraries are standard packages in all Linux distributions. Just open your favorite software package manager and install them.

   Package names may vary slightly in different distributions. See section Requirements for some hints on possible package names.

   As an example, commands to install the packages from command line in Ubuntu and Debian are provided, though you can do the same using synaptic, or aptitude. If you have another distribution, use your package manager to locate and install the appropriate library packages.

   Both in Debian and in Ubuntu you need to do:
   ```
   sudo apt-get install libboost-regex-dev libicu-dev zlib1g-dev
   sudo apt-get install libboost-system-dev libboost-program-options-dev
   ```

3. Download and Install FreeLing

   Download source package `freeling-4.0.tar.gz` from FreeLing webpage download section, and then execute:
   ```
   tar xzvf freeling-4.0.tar.gz
   cd freeling-4.0
   autoreconf --install
   ./configure
   make
   sudo make install
   ```

FreeLing library is entirely contained in the file `libfreeling.so` installed in `/usr/local/lib` by default.

Sample program `analyze` is installed in `/usr/local/bin`. See chapter about `analyzer` for details.

See `./configure --help` for options about installing in non-default directories or disabling some FreeLing options.

Install from `GitHub` repositories

Installing from GitHub is very similar to installing from source, but you'll have the chance to easily update your FreeLing to the latest development version.

1. Install development tools

   You'll need to install the C++ compiler, the GNU autotools, plus a `git` client.
   `sudo apt-get install build-essential automake autoconf libtool git`

   If you use a distribution different than Debian or Ubuntu, these packages may have different names. Use your package manager to locate and install the appropriate ones.

2. Install packaged requirements

   Follow the same procedure described in section about Source Installation for this step.

3. Checkout FreeLing sources

   If you want the latest development version, do:
   `git clone https://github.com/TALP-UPC/FreeLing.git mysrc`

   (you can replace `mysrc` with the directory name of your choice).

   If you want a previous release, after cloning the repository with the above command, you can checkout any previous tagged version with something like:
   `git checkout -b mybranch-v4 4.0-beta1`

   (that will create a new branch `mybranch-v4` in your local repository that will contain the version tagged as `4.0-beta1` in GitHub).

   You can find out the available tags at https://github.com/TALP-UPC/freeling.

   Versions older than 4.0 are available at the old SVN repository

4. Prepare local repositories for compilation
   `cd mysrc`
   `autoreconf --install`

5. Build and install FreeLing
   ```
   ./configure
   make
   sudo make install
   ```

If you keep the source directories, you will be able to update to newer versions at any moment with:

```
cd mysrc
git pull
./configure
make  sudo make install
```

Depending on what changed in the repository, you may need to issue `autoreconf --install` after `git pull`. You may also need to issue `make distclean` and repeat the process from `./configure` onwards.

Reducing needed disk space

FreeLing packages include linguistic data for all supported languages, which total up over 1Gb of disk space.

It is possible to safely remove data for languages that are not needed, saving that space.

- If installing from a `.deb` package, you can simply remove the unneeded language directories from `/usr/share/freeling/XX`. Make sure to keep `/usr/share/freeling/common`, `/usr/share/freeling/config`, and `/usr/share/freeling/XX` for any language XX you want to process.
- If installing from source (either from source package or from git repository) you can remove the unneeded data after installing, but it may be easier to remove it from source:
  - Remove the directories `data/XX` for any unneeded language. Make sure to keep directories `data/common`, `data/config`, and `data/XX` for any language XX you want to process.
  - After removing the unneeded directories, install normaly.

Locale-related problems when installing

If you get an error about `bad locale` when you enter `make install` or when you try to execute the `analyzer` sample program, you probably need to generate some locales in your system.

FreeLing uses `en_US.UTF8` locale as default during installation. If this locale is not installed in your system, you'll get an error during dictionary installation.

all languages in FreeLing should work with this locale, though Russian may need to have its own locale installed in the system.

The procedure to install a locale in your system varies depending on your distribution. For instance:

- In Ubuntu, you must use the `locale-get` command. E.g.:
  ```
  sudo locale-gen en_US.UTF8
  sudo locale-gen pt_BR.UTF8
  sudo locale-gen ru_RU.UTF8
  ...
  ```
- In Debian, you need to run the command:
  `dpkg-reconfigure locales`
  and select the desired locales from the list.

Installing on MacOS

Installing on MacOS is very similar to installing on Linux. The main difference is how to install the dependencies and required development tools, which is greatly eased by MacPorts.

- Install `XCode`.
  - Download and install `XCode` from Apple AppStore
  - Configure it with `sudo xcodebuild -license`
    `sudo xcode-select --install`
- Install `MacPorts`
  - Download and install `MacPorts` from [http://www.macports.org/install.php]
  - Update and configure:
    `sudo port -v selfupdate`
- Use MacPorts to install required developer tools:
  ```
  sudo port install automake
  sudo port install autoconf
  sudo port install libtool
  sudo port install git
  ```
- Use MacPorts to install required dependencies:
  `sudo port install boost`

  This will install also `libicu` and `zlib`. If configure complains about it not being there, you can install it with `sudo port install zlib`.

- Compile and install FreeLing using the procedure described either in install from source or in install from GitHub, but you must skip steps *"1. Install development tools"* and *"2. Install packaged requirements"* in that section and start directly at step 3.

  Important: libraries in MacOS are installed in `/opt/local` instead of `/usr/local`. So, when running `configure` command in step 3, you need to specify the right library paths, running `./configure` with the command:
  `./configure CPPFLAGS="-I/opt/local/include" LDFLAGS="-L/opt/local/lib"`

  You can add to this command any extra options you wish (`-enable-traces` , `-prefix` , etc). Use `./configure --help` to find out available options.

Executing

FreeLing is a library, which means that it not a final-user oriented executable program but a tool to develop new programs that require linguistic analysis services.

Nevertheless, a sample main program is included in the package for those who just want a text analyzer. This program may be adapted to fit your needs up to certain point (e.g. customized input/output formats, desired level of analysis) but very specific adaptations will require that you write your own main program, or adapt one of the existing examples.

The usage and options of this main program is described in this chapter.

Please take into account that this program is only a friendly interface to demonstrate FreeLing abilities, but that there are many other potential usages of FreeLing.

Thus, the question is not *why this program doesn't offer functionality X?*, *why it doesn't output information Y?*, or *why it doesn't present results in format Z?*, but *How should I use FreeLing library to write a program that does exactly what I need?*.

In the directory `src/main/simple_examples` in the tarball, you can find simpler sample programs that illustrate how to call the library, and that can be used as a starting point to develop your own application.

Porting to other platforms

FreeLing library is entirely written in C++, so it is possible to compile it on non-unix platforms with a reasonable effort.

FreeLing can be successfully built for MacOS, as described above.

It can also be built for MS-Windows using project files included in the tarball. You'll find the solution/project files and a README in the `msvc` folder inside FreeLing tarball. Binary packages for Windows can be found in GitHub FreeLing Releases page.

You can visit the Forum in FreeLing webpage for further help and details.

Linguistic Information Storage Classes

FreeLing library processes text and creates data structures that represent the linguistic objects in that text. Linguistic objects are elements like word, PoS-tag, sentence, parse_tree, etc.

Thus, FreeLing processing modules are able to convert a text into a collection of linguistic objects.

Processing modules typically receive some of these objects (e.g. a sentence, consisting of a list of words) and enrich it with additional information (e.g. adding PoS-tags to the words in the sentece, or a parse_tree to the sentence). Processing modules are described in section Language Processing Modules.

This chapter describes the classes where all this linguistic information is stored by the processing modules. The application calling FreeLing can later access this data structures to retrieve the results of the analyzers.

Classes `word` and `analysis`

The basic bricks of language are words, and this is the basic class that FreeLing uses to represent data.

Class `word` can store the form of a word (so, how it was written in the text), its phonetic encoding, and other information about it.

Many words are ambiguous with respect to their Part-of-Speech. For instance, the English word *bit* may be either a noun (a small piece of something) or a verb (past tense of verb *to bite*). Often, these different PoS tags are associated to different lemmas (e.g. *bit* for the noun reading and *bite* for the verb interpretation).

Thus, class `word` also stores a list of `analysis` objects. Every `analysis` is basically a combination of one lemma and one PoS tag. Analysis also may contain other information such as a list of senses, a probability, etc.

The word also stores information about which analysis were selected by the tagger, whether the word is a multiword made by the agglutination of several input tokens, etc. It also offers iterators to traverse the list of analysis of the word, either all of them, or only the subset selected (or non-selected) by the tagger.

The API for the class word is:

```
class word : public std::list<analysis> {

  public:
    /// user-managed data, we just store it.
    std::vector<std::wstring> user;

    /// constructor
    word();
    /// constructor
    word(const std::wstring &);
    /// constructor
    word(const std::wstring &, const std::list<word> &);
    /// constructor
    word(const std::wstring &, const std::list<analysis> &,
const std::list<word> &);
    /// Copy constructor
    word(const word &);
    /// assignment
    word& operator=(const word&);

    /// copy analysis from another word
    void copy_analysis(const word &);
    /// Get the number of selected analysis
    int get_n_selected(int k=0) const;
    /// get the number of unselected analysis
    int get_n_unselected(int k=0) const;
    /// true iff the word is a multiword compound
    bool is_multiword() const;
    /// true iff the word is a multiword marked as ambiguous
    bool is_ambiguous_mw() const;
    /// set mw ambiguity status
    void set_ambiguous_mw(bool);
    /// get number of words in compound
    int get_n_words_mw() const;
    /// get word objects that form the multiword
    const std::list<word>& get_words_mw() const;
    /// get word form
    const std::wstring& get_form() const;
    /// Get word form, lowercased.
    const std::wstring& get_lc_form() const;
```

```
    /// Get word phonetic form
    const std::wstring& get_ph_form() const;
    /// Get an iterator to the first selected analysis
    word::iterator selected_begin(int k=0);
    /// Get an iterator to the first selected analysis
    word::const_iterator selected_begin(int k=0) const;
    /// Get an iterator to the end of selected analysis list
    word::iterator selected_end(int k=0);
    /// Get an iterator to the end of selected analysis list
    word::const_iterator selected_end(int k=0) const;
    /// Get an iterator to the first unselected analysis
    word::iterator unselected_begin(int k=0);
    /// Get an iterator to the first unselected analysis
    word::const_iterator unselected_begin(int k=0) const;
    /// Get an iterator to the end of unselected analysis list
    word::iterator unselected_end(int k=0);
    /// Get an iterator to the end of unselected analysis list
    word::const_iterator unselected_end(int k=0) const;
    /// Get how many kbest tags the word has
    unsigned int num_kbest() const;
    /// get lemma for the selected analysis in list
    const std::wstring& get_lemma(int k=0) const;
    /// get tag for the selected analysis
    const std::wstring& get_tag(int k=0) const;

    /// get sense list for the selected analysis
    const std::list<std::pair<std::wstring,double> >&
get_senses(int k=0) const;
    std::list<std::pair<std::wstring,double> >& get_senses(int
k=0);
    // useful for java API
    std::wstring get_senses_string(int k=0) const;
    /// set sense list for the selected analysis
    void set_senses(const
std::list<std::pair<std::wstring,double> > &, int k=0);

    /// get token span.
    unsigned long get_span_start() const;
    unsigned long get_span_finish() const;
```

```
    /// get in_dict
    bool found_in_dict() const;
    /// set in_dict
    void set_found_in_dict(bool);
    /// check if there is any retokenizable analysis
    bool has_retokenizable() const;
    /// mark word as having definitive analysis
    void lock_analysis();
    /// check if word is marked as having definitive analysis
    bool is_locked() const;

    /// add an alternative to the alternatives list
    void add_alternative(const std::wstring &, int);
    /// replace alternatives list with list given
    void set_alternatives(const
std::list<std::pair<std::wstring,int> > &);
    /// clear alternatives list
    void clear_alternatives();
    /// find out if the speller checked alternatives
    bool has_alternatives() const;
    /// get alternatives list const &
    const std::list<std::pair<std::wstring,int> >&
get_alternatives() const;
    /// get alternatives list &
    std::list<std::pair<std::wstring,int> >& get_alternatives();
    /// get alternatives begin iterator
    std::list<std::pair<std::wstring,int> >::iterator
alternatives_begin();
    /// get alternatives end iterator
    std::list<std::pair<std::wstring,int> >::iterator
alternatives_end();
    /// get alternatives begin iterator
    std::list<std::pair<std::wstring,int> >::const_iterator
alternatives_begin() const;
    /// get alternatives end iterator
    std::list<std::pair<std::wstring,int> >::const_iterator
alternatives_end() const;

    /// add one analysis to current analysis list  (no duplicate
check!)
```

```
    void add_analysis(const analysis &);
    /// set analysis list to one single analysis, overwriting
current values
    void set_analysis(const analysis &);
    /// set analysis list, overwriting current values
    void set_analysis(const std::list<analysis> &);
    /// set word form
    void set_form(const std::wstring &);
    /// Set word phonetic form
    void set_ph_form(const std::wstring &);
    /// set token span
    void set_span(unsigned long, unsigned long);

    // get/set word position
    void set_position(size_t);
    size_t get_position() const;

    /// look for an analysis with a tag matching given regexp
    bool find_tag_match(const freeling::regexp &) const;

    /// get number of analysis in current list
    int get_n_analysis() const;
    /// empty the list of selected analysis
    void unselect_all_analysis(int k=0);
    /// mark all analysisi as selected
    void select_all_analysis(int k=0);
    /// add the given analysis to selected list.
    void select_analysis(word::iterator, int k=0);
    /// remove the given analysis from selected list.
    void unselect_analysis(word::iterator, int k=0);
    /// get list of analysis (useful for perl API)
    std::list<analysis> get_analysis() const;
    /// get begin iterator to analysis list (useful for
perl/java API)
    word::iterator analysis_begin();
    word::const_iterator analysis_begin() const;
    /// get end iterator to analysis list (useful for perl/java
API)
    word::iterator analysis_end();
    word::const_iterator analysis_end() const;
```

```
  };
```

Since a large amount of useful information about the word is stored in the analysis it contains, the class `analysis` also offers methods to access its content:

```
class analysis {

  public:
    /// user-managed data, we just store it.
    std::vector<std::wstring> user;

    /// constructor
    analysis();
    /// constructor
    analysis(const std::wstring &, const std::wstring &);
    /// assignment
    analysis& operator=(const analysis&);

    void init(const std::wstring &l, const std::wstring &t);
    void set_lemma(const std::wstring &);
    void set_tag(const std::wstring &);
    void set_prob(double);
    void set_distance(double);
    void set_retokenizable(const std::list<word> &);

    bool has_prob() const;
    bool has_distance() const;
    const std::wstring& get_lemma() const;
    const std::wstring& get_tag() const;
    double get_prob() const;
    double get_distance() const;
    bool is_retokenizable() const;
    std::list<word>& get_retokenizable();
    const std::list<word>& get_retokenizable() const;

    const std::list<std::pair<std::wstring,double> > &
get_senses() const;
    std::list<std::pair<std::wstring,double> > & get_senses();
    void set_senses(const
std::list<std::pair<std::wstring,double> > &);
```

```
    // useful for java API
    std::wstring get_senses_string() const;

    // get the largest kbest sequence index the analysis is
selected in.
    int max_kbest() const;
    // find out whether the analysis is selected in the tagger
k-th best sequence
    bool is_selected(int k=0) const;
    // mark this analysis as selected in k-th best sequence
    void mark_selected(int k=0);
    // unmark this analysis as selected in k-th best sequence
    void unmark_selected(int k=0);
  };
```

Class sentence

Words are grouped to form sentences. A `sentence` is basically a list of `word` objects, but it also may contain additional information, such as a parse tree, a dependency tree, or a list of predicates and arguments.

```
class sentence : public std::list<word> {

  public:
    sentence();
    ~sentence();
    sentence(const std::list<word>&);
    /// Copy constructor
    sentence(const sentence &);
    /// assignment
    sentence& operator=(const sentence&);
    /// positional access to a word
    const word& operator[](size_t) const;
    word& operator[](size_t);
    /// find out how many kbest sequences the tagger computed
    unsigned int num_kbest() const;
    /// add a word to the sentence
    void push_back(const word &);
```

```
    /// rebuild word positional index
    void rebuild_word_index();

    void clear();

    void set_sentence_id(const std::wstring &);
    const std::wstring& get_sentence_id() const;

    void set_is_tagged(bool);
    bool is_tagged() const;

    void set_parse_tree(const parse_tree &, int k=0);
    parse_tree & get_parse_tree(int k=0);
    const parse_tree & get_parse_tree(int k=0) const;
    bool is_parsed() const;

    void set_dep_tree(const dep_tree &, int k=0);
    dep_tree & get_dep_tree(int k=0);
    const dep_tree & get_dep_tree(int k=0) const;
    bool is_dep_parsed() const;

    /// get word list (useful for perl API)
    std::vector<word> get_words() const;
    /// get iterators to word list (useful for perl/java API)
    sentence::iterator words_begin();
    sentence::const_iterator words_begin() const;
    sentence::iterator words_end();
    sentence::const_iterator words_end() const;

    // obtain iterator to a word given its position
    sentence::iterator get_word_iterator(const word &w);
    sentence::const_iterator get_word_iterator(const word &w)
const;

    void add_predicate(const predicate &pr);
    /// see if word in given position is a predicate
    bool is_predicate(int p) const;
    /// get predicate number n for word in position p
    int get_predicate_number(int p) const;
    /// get word position for predicate number n
```

```
    int get_predicate_position(int n) const;
    /// get predicate for word at position p
    const predicate& get_predicate_by_pos(int n) const;
    /// get predicate number n
    const predicate& get_predicate_by_number(int n) const;
    /// get predicates of the sentence (e.g. to iterate over
them)
    const predicates & get_predicates() const;
  };
```

Trees stored in the sentence (either constituency trees or dependency trees) are a STL-like container that offers STL-like iterators to traverse the tree.

Predicates contain the word that heads the predicate in the sentence (typically a verb), plus a vector of arguments (other words in the sentence that play roles in that predicate).

Please check FreeLing Technical Reference Manual to find out details about the API for these classes.

Classes `paragraph` and `document`

Sentences can be grouped in `paragraphs`, and these can be grouped in a `document`

```
class paragraph : public std::list<sentence> {
  public:
    paragraph();
    paragraph(const std::list<sentence> &x);
    void set_paragraph_id(const std::wstring &);
    const std::wstring & get_paragraph_id() const;
  };
```

Class `document` is able to contain a sequence of `paragraphs`, each containing several `sentences`, each of which is made of `words` that have different `analysis`.

In addition, a `document` may contain also information about the coreference of different mentions in the text referring to the same entity, and provides methods to access this information.

Please check FreeLing Technical Reference Manual to find out details about the API for class `mention`.

```
class document : public std::list<paragraph> {

  public:
    document();
    ~document();

    bool is_parsed() const;
    bool is_dep_parsed() const;

    /// Adds one mention to the doc (the mention should be
already
    /// assigned to  a group of coreferents)
    void add_mention(const mention &m);

    // count number of words in the document
    int get_num_words() const;

    /// Gets the number of coreference groups found
    int get_num_groups() const;
    /// get list of ids of existing groups
    const std::list<int> & get_groups() const;

    /// Gets an iterator pointing to the beginning of the
mentions
    std::vector<mention>::const_iterator begin_mentions() const;
    std::vector<mention>::iterator begin_mentions();
    /// Gets an iterator pointing to the end of the mentions
    std::vector<mention>::const_iterator end_mentions() const;
    std::vector<mention>::iterator end_mentions();

    /// get reference to i-th mention
    const mention& get_mention(int) const;

    /// Gets all the nodes in a coreference group
    ///std::list<int> get_coref_nodes(int) const;

    /// Gets all the mentions' ids in the ith coreference group
found
    std::list<int> get_coref_id_mentions(int) const;
```

```
    /// Returns if two nodes are in the same coreference group
    /// bool is_coref(const std::wstring &, const std::wstring
&) const;

    const semgraph::semantic_graph & get_semantic_graph() const;
    semgraph::semantic_graph & get_semantic_graph();
  };
```

Finally, the `document` may contain also a semantic graph describing relations between entities and events.

The graph contains instances of `SG_entity` (an entity mentioned in the text, one or more times) and `SG_frame` (an event, relation or action described in the text). Objects of class `SG_frame` contain `SG_argument` instances that describe which entities are involved in that event.

Please check FreeLing Technical Reference Manual to find out details about the API for classes contained in the `semgraph`.

```
class semantic_graph {
    public:
      semantic_graph();
      ~semantic_graph();

      std::wstring add_entity(SG_entity &ent);
      std::wstring add_frame(SG_frame &fr);

      const SG_frame & get_frame(const std::wstring &fid) const;
      SG_frame & get_frame(const std::wstring &fid);

      std::wstring get_entity_id_by_mention(const std::wstring
&sid,
                                            const std::wstring
&wid) const;
      std::wstring get_entity_id_by_lemma(const std::wstring
&lemma,
                                          const std::wstring
&sens=L"") const;
      SG_entity & get_entity(const std::wstring &eid);
      const SG_entity & get_entity(const std::wstring &eid)
const;
```

```
    const std::vector<SG_entity> & get_entities() const;
    std::vector<SG_entity> & get_entities();

    const std::vector<SG_frame> & get_frames() const;
    std::vector<SG_frame> & get_frames();

    void add_mention_to_entity(const std::wstring &eid,
                               const SG_mention &m);
    void add_argument_to_frame(const std::wstring &fid,
                               const std::wstring &role,
                               const std::wstring &eid);

    bool is_argument(const std::wstring &eid) const;
    bool has_arguments(const std::wstring &fid) const;

    bool empty() const;
};
```

Language Processing Modules

This chapter describes each of the modules in FreeLing. For each module, its public API is described, as well as its main configuration options. Most modules can be customized via a configuration file.

A typical module receives a list of sentences, and enriches them with new linguistic information (morphological analysis, disambiguation, parsing, etc.)

Usually, when the module is instantiated, it receives as a parameter the name of a file where the data and/or parameters needed by the module are stored (e.g. a dictionary file for the dictionary search module, or a CFG grammar for a parser module).

Most modules are language-independent, that is, if the provided file contains data for another language, the same module will be able to process that language.

All modules are thread-safe (though some such as the sentence splitter need a session identifier token to ensure this safety).

If an application needs to process more than one language, it can instantiate the needed modules for each language simply calling the constructors with different data files as a parameter.

Main processing classes in FreeLing

- Part-of-Speech Tagger
- Phonetic Encoding
- Named Entity Classification
- Chart Parser
- Rule-based Dependency Parser
- Statistical Dependency Parser and SRL
- Coreference Resolution
- Semantic Graph Extraction

Other Useful Modules

FreeLing contains some internal classes used by the analysis modules described above, as well as other utility classes (e.g. to handle input/output formats). Some applications may be interested in directly accessing these lower-level utilities, the most relevant of which are:

- Analyzer Metamodule
- Tagset Management
- Semantic Database
- Approximate Search Dictionary
- Feature Extractor
- Input/Output Formats

Language Identifier Module

This module is somehow different of the other modules, since it doesn't enrich the given text. It compares the given text with available models for different languages, and returns the most likely language the text is written in. It can be used as a preprocess to determine which data files are to be used to analyze the text.

The API of the language identifier is the following:

```
class lang_ident {
  public:
    /// Build an empty language identifier.
    lang_ident();

    /// Build a language identifier, read options from given
file.
    lang_ident(const std::wstring &cfgfile);

    /// load given language from given model file, add to
existing languages.
    void add_language(const std::wstring &modelfile);

    /// train a n-gram model of given order for language 'code',
    /// store in modelFile, and add it to the known languages
list.
    void train_language(const std::wstring &textfile,
                        const std::wstring &modelfile,
                        const std::wstring &code, size_t order);

    /// Classify the input text and return the code of the best
language among
    /// those in given set. If set is empty all known languages
are considered.
    /// If no language reaches the threshold,  "none" is
returned
    std::wstring identify_language (
                    const std::wstring&,
                    const std::set<std::wstring> &ls) const;
```

```
    /// Classify the input text and return the code and
perplexity for
    /// each language in given set. If set is empty, all known
languages
    /// are considered.
    void rank_languages (
                std::vector<std::pair<double,std::wstring> >
&result,
                const std::wstring &text,
                const std::set<std::wstring> &ls) const;


};
```

Once created, the language identifier may be used to get the most likely language of a text using the method `identify_language`, or to return a sorted vector of probabilities for each language (`rank_languages`). In both cases, a set of languages to be considered may be supplied, telling the identifier to apply to the input text only models for those languages in the list. An empty list is interpreted as *use all available language models*. The language list parameter is optional in both identification methods, and defaults to the empty list.

The same `lang_ident` class may be used to train models for new languages. A plain text file can be passed to method `train_language` which will use the data to create a new model, which will enlarge the identifier's language repertoire, and will be stored for its use in future instances of the class.

The constructor expects a configuration file name, containing information about where are the language models located, and some parameters. The contents of that file are described below.

Language Identifier Options File

The language identifier options file has a unique section `<Languages>` closed by tag `</Languages>` .

Section `<Languages>` contains a list of filenames, one per line. Each filename contains a language model (generated with the `train_language` method). The filenames may be absolute or relative. If relative, they are considered to be relative to the location of the identifier options file.

An example of a language identifier option file is:

```
<Languages>
./es.dat
./ca.dat
./it.dat
./pt.dat
</Languages>
```

Tokenizer Module

The first module in the processing chain is the tokenizer. It converts plain text to a vector of `word` objects, according to a set of tokenization rules.

Tokenization rules are regular expressions that are matched against the beggining of the text line being processed. The first matching rule found is used to extract the token, the matching substring is deleted from the line, and the process is repeated until the line is empty.

The API of the tokenizer module is the following:

```
class tokenizer {
  public:
    /// Constructor
    tokenizer(const std::wstring &cfgfile);

    /// tokenize string, leave result in lw
    void tokenize(const std::wstring &text,
                  std::list<word> &lw) const;

   /// tokenize string, return result as list
    std::list<word> tokenize(const std::wstring &text) const;

    /// tokenize string, updating byte offset. Leave results in
lw.
    void tokenize(const std::wstring &text,
                  unsigned long &offset,
                  std::list<word> &lw) const;

    /// tokenize string, updating offset, return result as list
    std::list<word> tokenize(const std::wstring &text,
                             unsigned long &offset) const;
};
```

That is, once created, the tokenizer module receives plain text in a string, tokenizes it, and returns a list of `word` objects corresponding to the created tokens

Tokenizer Rules File

The tokenizer rules file is divided in three sections `<Macros>`, `<RegExps>` and `<Abbreviations>`. Each section is closed by `</Macros>`, `</RegExps>` and `</Abbreviations>` tags respectively.

The `<Macros>` section allows the user to define regexp macros that will be used later in the rules. Macros are defined with a name and a POSIX regexp. E.g.:

```
MYALPHA [A-Za-z]
ALPHA [[:alpha:]]
```

The `<RegExps>` section defines the tokenization rules. Previously defined macros may be referred to with their name in curly brackets. E.g.:

```
*ABREVIATIONS1 0 ((\{ALPHA\}+\.)+)(?!\.\.)
```

Rules are regular expressions, and are applied in the order of definition. The first rule matching the beginning of the line is applied, a token is built, and the rest of the rules are ignored. The process is repeated until the line has been completely processed.

The format of each rule is:

- The first field in the rule is the rule name. If it starts with a `*`, the RegExp will only produce a token if the match is found in the abbreviation list (`<Abbreviations>` section). Apart from that, the rule name is only for informative/readability purposes.
- The second field in the rule is the substring to form the token/s with. It may be 0 (the match of the whole expression) or any number from 1 to the number of subexpression (up to 9). A token will be created for each subexpression from 1 to the specified value.
- The third field is the regexp to match against the input. line. Any POSIX regexp convention may be used.
- An optional fourth field may be added, containing the string `CI` (standing for Case Insensitive). In this case, the input text will be matched case-insensitively against the regexp. If the fourth field is not present, or it is different than `CI`, the rule is matched case-sensitively.

The `<Abbreviations>` section defines common abbreviations (one per line) that must not be separated of their following dot (e.g. `etc.`, `mrs.`). They must be lowercased, even if they are expected to appear uppercased in the text.

Splitter Module

The splitter module receives lists of `word` objects (either produced by the tokenizer or by any other means in the calling application) and buffers them until a sentence boundary is detected. Then, a list of `sentence` objects is returned.

The buffer of the splitter may retain part of the tokens if the given list didn't end with a clear sentence boundary. The caller application can submit further token lists to be added, or request the splitter to flush the buffer.

In order to be thread-safe, the splitter needs to keep different internal buffers for different threads (or for the same thread, if it uses the same splitter to split two different text sources alternatively). Thus, the calling application must open a splitting session to obtain an id, that will identify the internal buffer to use.

In addition, each session will have its own sentence-id counter, so sentences processed in the same session are assigned correlative sentence-ids (starting at 1).

The API for the splitter class is:

```
class splitter {
  public:
    /// Constructor. Receives a file with the desired options
    splitter(const std::wstring &cfgfile);

    /// Destructor
    ~splitter();

    /// splitter::session_id is the type to store session ids
    typedef int session_id;

    /// open a splitting session, get session id
    session_id open_session();

    /// close splitting session
    void close_session(session_id ses);

    /// Add given list of words to the buffer, and put complete
sentences
    /// that can be build into ls.
    /// The boolean states if a buffer flush has to be forced
(true) or
    /// some words may remain in the buffer (false) if the
splitter
    /// needs to wait to see what is coming next.
    /// Each thread using the same splitter needs to open a new
session.
    void split(session_id ses,
               const std::list<word> &lw,
               bool flush,
               std::list<sentence> &ls);

    /// same than previous method, but result sentences are
returned.
    std::list<sentence> split(session_id ses,
                              const std::list<word> &lw,
                              bool flush);

};
```

Splitter Options File

The splitter options file contains four sections: `<General>`, `<SentenceEnd>`, `<SentenceStart>`, and `<Markers>`.

The `<General>` section contains general options for the splitter: Namely, `AllowBetweenMarkers` and `MaxWords` options. The former may take values 1 or 0 (on/off). The later may be any integer. An example of the `<General>` section is:

```
<General>
AllowBetweenMarkers 0
MaxWords 0
</General>
```

If `AllowBetweenMarkers` is off (`0`), a sentence split will never be introduced inside a pair of parenthesis-like markers, which is useful to prevent splitting in sentences such as *"I hate --Mary said. Angryly.-- apple pie"*. If this option is on (`1`), sentence splits will be introduced as if they had happened outside the markers.

The value for `MaxWords` states how many words are processed before forcing a sentence split inside parenthesis-like markers (this option is intended to avoid memory fillups in case the markers are not properly closed in the text).
A value of zero means *Never split, I'll risk to a memory fillup*. This option is less aggressive than unconditionally activating `AllowBetweenMarkers`, since it will introduce a sentence split between markers only after a sentence of length `MaxWords` has been accumulated. Setting `MaxWords` to a large value will prevent memory fillups, while keeping at a minimum the splittings inside markers.

The `<Markers>` section lists the pairs of characters (or character groups) that have to be considered open-close markers. For instance:

```
<Markers>
 " "
 ( )
 { }
 /* */
 </Markers>
```

The `<SentenceEnd>` section lists which characters are considered as possible sentence endings. Each character is followed by a binary value stating whether the character is an unambiguous sentence ending or not. For instance, in the following example, `?` is an unambiguous sentence marker, so a sentence split will be introduced unconditionally after each `?`. The other two characters are not unambiguous, so a sentence split will only be introduced if they are followed by a capitalized word or a sentence start character.

```
<SentenceEnd>
 . 0
 ? 1
 ! 0
 </SentenceEnd>
```

The `<SentenceStart>` section lists characters known to appear only at sentence beggining. For instance, open question/exclamation marks in Spanish:

```
<SentenceStart>
 ?
 !
 </SentenceStart>
```

Morphological Analyzer Module

The morphological analyzer is a meta-module which does not perform any processing of its own.

It is just a convenience module to simplify the instantiation and call to the submodules described in the next sections.

At instantiation time, it receives a `maco_options` object, containing information about which submodules have to be created and which files must be used to create them.

Any submodule loaded at instantiation time, may be deactivated/reactivated later using the method `maco::set_active_options` described above. Note that a submodule that was not loaded when creating a `maco` instance, can neither be loaded nor activated later.

A calling application may bypass this module and just call directly the submodules.

The Morphological Analyzer API is:

```
class maco {
  public:
    /// Constructor. Receives a set of options.
    maco(const maco_options &);

    /// change active options for further analysis
    void set_active_options(bool umap, bool num, bool pun, bool
dat,
                            bool dic, bool aff, bool comp, bool
rtk,
                            bool mw, bool ner, bool qt, bool
prb);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The `maco_options` class has the following API:

```
class maco_options {
  public:
    // Language analyzed
    std::wstring Lang;

    /// Morphological analyzer modules configuration/data files.
    std::wstring LocutionsFile, QuantitiesFile, AffixFile,
      CompoundFile, DictionaryFile, ProbabilityFile, NPdataFile,
      PunctuationFile, UserMapFile;
```

```
    /// module-specific parameters for number recognition
    std::wstring Decimal, Thousand;

    /// module-specific parameters for probabilities
    double ProbabilityThreshold;

    /// module-specific parameters for dictionary
    bool InverseDict,RetokContractions;

    /// constructor
    maco_options(const std::wstring &);

    /// Option setting methods provided to ease perl interface
generation.
    /// Since option data members are public and can be accessed
directly
    /// from C++, the following methods are not necessary, but
may become
    /// convenient sometimes.
    void set_data_files(const std::wstring &usr,
                        const std::wstring &pun, const
std::wstring &dic,
                        const std::wstring &aff, const
std::wstring &comp,
                        const std::wstring &loc, const
std::wstring &nps,
                        const std::wstring &qty, const
std::wstring &prb);
    void set_nummerical_points(const std::wstring &dec, const
std::wstring &tho);
    void set_threshold(double);
    void set_inverse_dict(bool);
    void set_retok_contractions(bool);
  };
```

To instantiate a Morphological Analyzer object, the calling application needs to create an instance of `maco_options`, initialize its fields with the desired values, and use it to call the constructor of the `maco` class.

Each possible submodule will be created and loaded if the given file is different than the empty string. The created object will create the required submodules, and when asked to `analyze` some sentences, it will just pass it down to each submodule, and return the final result.

Class `maco_options` has convenience methods to set the values of the options, but note that all the members are public, so the user application can set those values directly if preferred.

Submodules called by the `maco` class are the following. Note that any of them can be instantiated and used directly if needed.

- Punctuation Detection
- Number Detection
- User Map Module
- Dates Detection
- Dictionary Search
- Multiword Recognition
- Named Entity Recognition
- Quantity Recognition
- Probability Assignment and Guesser

Punctuation Detection Module

The punctuation detection module assigns Part-of-Speech tags to punctuation symbols. The API of the class is the following:

```
class punts {
  public:
    /// Constructor: load punctuation symbols and tags from
given file
    punts(const std::string &cfgfile);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor receives as parameter the name of a file containing the list of the PoS tags to be assigned to each punctuation symbol.

Note that this module will be applied afer the tokenizer, so, it will only annotate symbols that have been separated at the tokenization step. For instance, if you include the three suspensive dots (...) as a single punctuation symbol, it will have no effect unless the tokenizer has a rule that causes these substring to be tokenized in one piece.

Punctuation Tags File

The format of the file listing the PoS for punctuation symbols is one punctuation symbol per line, each line with the format: `punctuation-symbol lemma tag`. E.g.:

```
! ! Fat
, , Fc
: : Fd
... ... Fs
```

One special line may be included defining the tag that will be assigned to any other punctuation symbol not found in the list. Any token containing no alphanumeric character is considered a punctuation symbol. This special line has the format: `<Other> tag`.
E.g.:
`<Other> Fz`

Number Detection Module

The number detection module is language dependent: It recognizes nummerical expression (e.g.: `1,220.54` or `two-hundred sixty-five`), and assigns them a normalized value as lemma.

The module is basically an Augmented Transition Network (ATN) that recognizes valid nummerical expressions. Since the structure of the automata and the actions to compute the actual nummerical value are different for each lemma, the ATN is coded in C++ and has to be rewritten for any new language.

For languages that do not have an implementation of a specific automata, a generic module is used to recognize number-like expressions that contain nummerical digits.

There is no configuration file to be provided to the class when it is instantiated.

The API of the class is:

```
class numbers {
  public:
    /// Constructor: receives the language code, and the decimal
    /// and thousand point symbols
    numbers(const std::string &lang,
            const std::string &dec,
            const std::string &thou);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

Parameters expected by the constructor are:

- The language code: used to decide whether the generic recognizer or a language-specific module is used.
- The decimal point symbol.
- The thousand point sympol.

The last two parameters are needed because in some latin languages, the comma is used as decimal point separator, and the dot as thousand mark, while in languages like English it is the other way round. These parameters make it possible to specify what character is to be expected at each of these positions. They will usually be comma and dot, but any character could be used.

User Map Module

The user map module assigns Part-of-Speech tags to words matching a given regular expression. It can be used to customize the behaviour of the analysis chain to specific applications, or to process domain-specific special tokens.

If used, it should be run before the other modules (e.g. morphological analysis) to prevent them from guessing possible PoS tags for special tokens.

The API of the class is the following:

```
class RE_map {
  public:
    /// Constructor, load special token patterns
    RE_map(const std::wstring &cfgfile);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor receives as parameter the name of a file containing a list of regular expressions, and the list of pairs lemma-PoS tag to be assigned to each word matching the expression.

Note that this module will be applied afer the tokenizer, so, it will only annotate symbols that have been separated at the tokenization step. So, customizing your application to recognize certain special tokens may require modifying also the tokenizer configuration file.

Note also that if you introduce in this file PoS-tags which are not in the tagset known to the tagger, it may not be able to properly disambiguate the tag sequence.

Finally, note that this module sequentially checks each regular expression in the list against each word in the text. Thus, it should be used for patterns (not for fixed strings, which can be included in a dictionary file), and with moderation: using a very long list of expressions may severely slow down your analysis chain.

User Map File

The format of the file containing the user map from regular expression to pairs lemma-PoS is one regular expression per line, each line with the format: `regex lemma1 tag1 lemma2 tag2 ...`.

The lemma may be any string literal, or $$ meaning that the string matching the regular expression is to be used as a lemma. E.g.:

```
@[a-z][0-9] $ NP00000
<.*> XMLTAG Fz
hulabee hulaboo JJS hulaboo NNS
```

The first rule will recognize tokens such as `@john` or `@peter4`, and assign them the tag `NP00000` (proper noun) and the matching string as lemma.

The second rule will recognize tokens starting with `<` and ending with `>` (such as `<HTML>` or `<br/>`) and assign them the literal `XMLTAG` as lemma and the tag `Fz` (punctuation:others) as PoS.

The third rule will assign the two pairs lemma-tag to each occurrence of the word *hulabee*. This is just an example, and if you want to add a word to your dictionary, the dictionary module is the right place to do so.

Examples of possible configuration files for this module can be found in FreeLing data directory:

- `ca/twitter/usermap.dat`
- `es/twitter/usermap.dat`
- `en/twitter/usermap.dat`

Dates Detection Module

The dates detection module, as the number detection module, is a collection of language-specific Augmented Transition Networks, and for this reason needs no data file to be provided at instantiation time.

For languages that do not have a specific ATN, a default analyzer is used that detects simple date patterns (e.g. `DD-MM-AAAA`, `MM/DD/AAAA`, etc.)

The API of the class is:

```
class dates {
  public:
    /// Constructor: receives the language code
    dates(const std::string &lang);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The only parameter expected by the constructor is the language of the text to analyze, in order to be able to apply the appropriate specific automata, or select the default one if none is available.

The dates module will create a multiword for the whole date expression, and set a lemma for the multiword that normalizes the fields of the date/time expression. Fields not specified in the expression are set to `??`. E.g.:

- The expression *thursday, may 15th, two thousand and six* will generate the multiword `thursday_,_may_15th_,_two_thousand_and_six`, and set its lemma to

`[J:15/5/2002:??.??:??]`

- The expression *may 15th at ten past four in the afternoon* will generate the multiword `may_15th_at_ten_past_four_in_the_afternoon`, and set its lemma to `[??:15/5/??:4.10:pm]`

The lemma has the format `[w:DD/MM/YYYY:hh.mm:tt]`, where:

- `w`: Weekday code (`L`: Monday, `M`: Tuesday, `X`: Wednesday, `J`: Thursday, `V`: Friday, `S`: Saturday, `D`: Sunday)
- `DD`: day number
- `MM`: month number
- `YYYY`: year number
- `hh`: hour
- `mm`: minute
- `tt`: `am` or `pm`

Dictionary Search Module

The dictionary search module has two functions: Search the word forms in the dictionary to find out their lemmas and PoS tags, and apply affixation or compounding rules to find the same information in the cases in which the form is a derived form not included in the dictionary (e.g. the word *quickly* may not be in the dictionary, but a suffixation rule may state that removing suffix *-ly* and searching for the obtained adjective is a valid way to form and adverb).

The decision of what is included in the dictionary and what is dealt with through affixation rules is left to the linguist building the linguistic data resources.

The API for this module is the following:

```
class dictionary {
  public:
    /// Constructor
    dictionary(const std::wstring &Lang, const std::wstring
&dicFile,
               const std::wstring &sufFile, const std::wstring
&compFile,
               bool invDic=false, bool retok=true);

    /// Destructor
    ~dictionary();

    /// add analysis to dictionary entry (create entry if not
there)
    void add_analysis(const std::wstring &form, const analysis
&an);

    /// remove entry from dictionary
    void remove_entry(const std::wstring &form);

    /// customize behaviour of dictionary for further analysis
    void set_retokenize_contractions(bool b);
    void set_affix_analysis(bool b);
    void set_compound_analysis(bool b);
```

```
    /// find out whether the dictionary has loaded an affix or
compounds module
    bool has_affixes() const;
    bool has_compounds() const;

    /// Get dictionary entry for a given form, add to given
list.
    void search_form(const std::wstring &form,
std::list<analysis> &la) const;

    /// Fills the analysis list of a word, checking for suffixes
and contractions.
    /// Returns true iff the form is a contraction, returns
contraction components
    /// in given list
    bool annotate_word(word &w, std::list<word> &lw, bool
override=false) const;

    /// Fills the analysis list of a word, checking for suffixes
and contractions.
    /// Never retokenizing contractions, nor returning component
list.
    /// It is just a convenience equivalent to
"annotate_word(w,dummy,true)"
    void annotate_word(word &w) const;

    /// Get possible forms for a lemma+pos (requires
inverseDict=true
    /// when module was instantiated)
    std::list<std::wstring> get_forms(const std::wstring &lemma,
                                      const std::wstring &pos)
const;

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
```

```
    sentence analyze(const sentence &ls) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The parameters of the constructor are:

- The language of the processed text. This is required by the affixation submodule to properly handle graphical accent rules in latin languages.
- The dictionary file name. See below for details.
- The affixation rules file name (an empty string means no affix analysis is to be performed).
- The compound rules file name (an empty string means no compound analysis is to be performed).
- An optional boolean (default: false) stating whether the dictionary must be created with inverse access, to enable the use of `get_forms` to obtain a word form given a lemma+PoS pair.
- An optional boolean (default: true) stating whether the contractions found in the dictionary must be retokenized right away, or left for later modules to decide.

The dictionary content may be updated online using the methods `add_analysis` and `remove_entry`. Notice that this will modify the dictionary instance in RAM, but not the file in the disk. If you want your changes to be permanent, modify the dictionary file instead.

The dictionary module behaviour may be tuned on-the-fly with the methods `set_affix_analysis`, `set_retokenize_contractions`, and `set_compound_analysis`, which will activate (when parameter is `true`) or deactivate (when `false`) the specified feature for any further call to the `analyze` methods.

Form Dictionary File

The form dictionary contais two required sections: `<IndexType>` and `<Entries>`

Section `<IndexType>` contains a single line, that may be either `DB_PREFTREE` or `DB_MAP`. With `DB_MAP` the dictionary is stored in a C++ STL `map` container. With `DB_PREFTREE` it is stored in a prefix-tree structure. Depending on the size of the dictionary and on the morphological variation of the language, one structure may yield slightly better access times than the other.

Section `<Entries>` contains lines with one form per line. Each form line has format: `form lemma1 PoS1 lemma2 PoS2 ...`. E.g.:

```
casa casa NCFS000 casar VMIP3S0 casar VMM02S0
backs back NNS back VBZ
```

Lines corresponding to words that are contractions may have an alternative format if the contraction is to be splitted. The format is `form form1+form2+... PoS1+PoS2+...` where `form1,form2,...` are the forms (not the lemmas) of the contracted words. For instance:
`del de+el SP+DA`

This line expresses that whenever the form *del* is found, it is replaced with two words: *de* and *el*. Each of the new two word forms are searched in the dictionary, and assigned any tag matching their correspondig tag in the third field. So, *de* will be assigned all tags starting with `SP` that this entry may have in the dictionary, and *el* will get any tag starting with `DA`.

Note that a contraction included in the dictionary cannot be splitted in two different ways corresponding to different forms (e.g. `he's = he+is | he+has`), so only a combination of forms and a combination of tags may appear in the dictionary. That is, the dictionary entry:
`he's he+is PRP+VB he+has PRP+VB`
is invalid because it states two different forms for *'s*, while the entry:
`he's he+'s PRP+VB`
is valid because it only states a form (*'s*) that will get two lemmas (*be* and *have*) when looked up in the dictionary.

Also, a set of tags may be specified for a given form, e.g.:
`he'd he+'d PRP+VB/MD`

This will produce two words: *he* with `PRP` analysis, and *'d* with its analysis matching any of the two given tags (i.e. `have_VBZ` and `would_MD`). Note that this will work only if the form 'd is found in the dictionary with those possible analysis.

If all tags for one of the new forms are to be used, a wildcard may be written as a tag. e.g.:
`pal para+el SP+*`

This will replace pal with two words, para with only its `SP` analysis, plus el with all its possible tags.

The contraction entries in the dictionary are intended for inambiguous contractions, or for cases such that it is not worth (or it is too difficult) to handle otherwise. For splitting more sophisticated compound words, such as verb clitic pronouns in Spanish or Italian (e.g dale → dar+él), derivation (e.g. *quick* → *quickly*, rápida → rápidamente), diminutive/augmentative sufixes, prefixes, or other similar behaviours, the affixation module should be used.

An optional parameter in the constructor enables to control whether contractions are splitted by the dictionary module itself (thus passing two words instead of one to later modules) or the decision is left to later modules (which will receive a single word carrying retokenization information).

Affixation Rules File

The submodule of the dictionary handler that deals with affixes requires a set of affixation rules.

The file consists of two (optional) sections: `<Suffixes>` and `<Prefixes>`. The first one contains suffixation rules, and the second, prefixation rules. They may appear in any order.

Both kinds of rules have the same format, and only differ in whether the affix is checked at the beggining or at the end of the word.

Each rule has to be written in a different line, and has 10 fields:

1. Affix to erase form word form (e.g: *crucecita* - *cecita* = *cru*)
2. Affix (`*` for emtpy string) to add to the resulting root to rebuild the lemma that must be searched in dictionary (e.g. *cru* + *z* = *cruz*)
3. Condition on the tag of found dictionary entry (e.g. *cruz* has tag `NCFS` in the dictionary). The condition is a perl RegExp
4. Tag for suffixed word (`*` = keep tag in dictionary entry)
5. Check lemma adding accents
6. Enclitic suffix (special accent behaviour in Spanish)
7. Prevent later modules (e.g. probabilities) from assigning additional tags to the word
8. Lemma to assign: Any combination of: `F`, `R`, `L`, `A`, or a string literal separated with a + sign. For instance: `R+A`, `A+L`, `R+mente`, etc.

   `F` stands for the original form (before affix removal, e.g. *crucecitas*), `R` stands for root found in dictionary (after affix removal and root reconstruction, e.g. *cruces*), `L` stands for lemma in matching dictionary entry (e.g. *cruz*), `A` stands for the affix that the rule removed

9. Try the affix always, not only for unknown words.
10. Retokenization info, explained below (" `-` " for none)

Example of prefix rule:

```
anti * ^NC AQ0CN0 0 0 1 A+L 0 -
```

This prefix rule states that `anti` should be removed from the beggining of the word, nothing (`*`) should be added, and the resulting root should be found in the dictionary with a `NC` PoS tag. If that is satisfied, the word will receive the `AQ0CN0` tag and its lemma will be set to the affix (`anti`) plus the lemma of the root found in the dictionary. For instance, the word `antimisiles` would match this rule: `misiles` would be found in the dictionary with lema `misil` and PoS `NCMP000`. Then, the word will be assigned the lemma `antimisil` (`A+L = anti+misil`) and the tag `AQ0CN0`.

Examples of sufix rules:

```
cecita z|za ^NCFS NCFS00A 0 0 1 L 0 -
les * ^V * 0 1 0 L 1 $$+les:$$+PP
```

The first suffix rule above (`cecita`) states a suffix rule that will be applied to unknown words, to see whether a valid feminine singular noun is obtained when substituting the suffix `cecita` with `z` or `za`. This is the case of `crucecita` (diminutive of `cruz`). If such a base form is found, the original word is analyzed as diminutive suffixed form. No retokenization is performed.

The second rule (`les`) applies to all words and tries to check whether a valid verb form is obtained when removing the suffix `les`. This is the case of words such as `viles` (which may mean I saw them, but also is the plural of the adjective `vil`).

In this case, the retokenization info states that if eventually the verb tag is selected for this word, it may be retokenized in two words: The base verb form (referred to as `$$`, `vi` in the example) plus the word `les`. The tags for these new words are expressed after the colon: The base form must keep its PoS tag (this is what the second `$$` means) and the second word may take any tag starting with PP the word may have in the dictionary. Prefixing a tag in the retokenization information with a `!` sign will assign that tag without performing any dictionary check. (E.g. if the rule was `$$+les:$$+!XYZ`, the word `les` would get the tag `XYZ` regardless of whether either the word or the tag are found in the dictionary.

So, for word `viles`, we would obtain its adjective analysis from the dictionary, plus its verb + clitic pronoun from the suffix rule: `viles vil AQ0CP0 ver+les VMIS1S0+PP3CD00`

The second analysis will carry the retokenization information, so if eventually the PoS tagger selects the `VMI` analysis (and the TaggerRetokenize option is set), the word will be retokenized into:

```
vi ver VMIS1S0
les ellos PP3CPD00
```

Compound rules file

The dictionary may be configured to check whether a word is a compound formed by the concatenation of several dictionary words.

The compounds will be detected if they are formed strictly by words in the dictionary, either glued toghether (e.g. *ghostbusters*, *underworld*), or separated with a character especified to be a compound joint (e.g. a dash: *middle-aged*, self-sustained*). In the current version, if a component of the compound is a form derived via affixation (e.g. _extremely-complex*) it will not be detected, even if the base form for the affixed word (*extreme* in this case) is in the dictionary.

The submodule of the dictionary handler that deals with compounds requires a set of valid compound patterns. The configuration file for the compound detection module consists of three sections: `<UnknownWordsOnly>`, `<JoinChars>`, and `<CompoundPatterns>`.

Section `<UnknownWordsOnly>` is optional, and contains just one line with either `yes` or `no`. Default is `yes`, meaning that only words not found in the dictionary or solved by the affixation module will be tried as potential compounds. If the option is set to `no`, all words will be tried as potential compounds (probably overgenerating wrong analysis).

Section `<JoinChars>` contains a list of characters to be considered compound joints. A typical member of this list is the dash ``-'', though the list may vary from one language to another. The character list must be specified with one character per line.

Section `<CompoundPatterns>` contains a list of valid compound patterns. Each pattern must be defined in a separate line.

There are two possible formats for specifying a pattern:

- `headed_PoS_pattern` This pattern consists of a sequence of PoS tag prefixes separated by underscores. Exactly one of the tags must be preceeded with a plus sign (`+`) that indicates it is the head of the compound, and thus its PoS tag will be used as the tag of the compound.

  For instance, the pattern `JJ_+NN` will recognize compounds such as *lightweight* or *lightweights*, since their components match the categories in the pattern. The compound will receive the PoS tag of the second component, producing the analyses `(light_weight,NN)` and `(light_weight,NNS)` respectively.

- `PoS_pattern compound_PoS` Patterns specified in this way behave like described above, but no head marker is expected in the `PoS_pattern`. Instead of using the head PoS, the PoS tag for the compound is straightforwardly set to the value of `compound_PoS`.

For flexibility, in any of both rule formats, a wildcard `*` may be used instead of a PoS prefix, causing that part of the pattern to match any PoS tag.

Dictionary Management

In many NLP applications you want to deal with text in a particular domain, which contain words or expressions (terminology, proper nouns, etc.) that are specific or have a particular meaning in that domain.

Thus, you may want to extend you dictionary to include such words. There are two main ways of doing this:

- Extend your dictionary: Dictionaries are created at installation time from sources in the directory `data/XX/dictionary` (where `XX` is the language code). Those files contain one triplet `word lemma tag` per line, and are fused in a single dictionary at installation time.

  The script in `src/utilities/dicc-management/bin/build-dict.sh` will read the given files and build a dictionary with all them (check the file `src/utilities/dicc-management/README` for details).

  Thus, if you have a domain dictionary with a list of triplets `word lemma tag`, you can build a new dictionary fusing the original FreeLing entries with your domain entries. The resulting file can be given to the constructor of the dictionary class.

- Instantiate more than one dictionary module: Another option is to instatiate several dictionary modules (creating each one with a different dictionary file), and run the text through each of them sequentially. When the word is found in one dictionary along the chain, the following dictionary modules will ignore it and will not attempt to look it up.

Multiword Recognition Module

This module aggregates input tokens in a single word object if they are found in a given list of multiwords.

The API for this class is:

```
class locutions: public automat {
  public:
    /// Constructor, receives the name of the file
    /// containing the multiwords to recognize.
    locutions(const std::string &cfgfile);

    /// Detect multiwords starting at given sentence position
    bool matching(sentence &s, sentence::iterator &p);

    /// set "OnlySelected" flag on/off
    void set_OnlySelected(bool b);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

Class `automat` implements a generic ATN. The `locutions` class is a derived class which implements an ATN to recognize the word patterns listed in the file given to the constructor.

Multiword Definition File

The file contains a list of multiwords to be recognized.

The file consists of three sections `<TagSetFile>`, `<OnlySelected>`, and `<Multiwords>`. All of them are optional (though an empty or unexisting `<Multiwords>` section will result in never detecting any multiword).

Section `<TagSetFile>`. This section contains a single line with the path to a tagset description file to be used when computing short versions for PoS tags. If the path is relative, the location of the multiwords file is used as the base directory.

Section `<OnlySelected>` contains a single line with the one of the words `yes`, `true`, `no`, or `false`. If the section is ommited, or contains any unknown value, the value defaults to `false`.

This flag controls the analysis that the multiword detector will consider when matching a multiwords pattern that contains lemmas or PoS descriptions. If `OnlySelected=` is set to `false` (the default) all possible analysis for each word will matched against the pattern. If `OnlySelected` is set to `true`, only selected analysis will be checked. Note that setting `OnlySelected` to `true` only makes sense if the multiwords module is applied after the PoS tagger, since otherwise no analysis will be selected.

Section `<Multiwords>` contains one multiword pattern per line. Each line has the format: `form lemma1 pos1 lemma2 pos2 ... [A|I]`

The multiword form may contain lemmas in angle brackets, meaning that any form with that lemma will be considered a valid component for the multiword.

The form may also contain PoS tags. Any uppercase component in the form will be treated as a PoS tag.

Any number of pairs lemma-tag may be assigned to the multiword. The PoS tagger will select the most probable given the context, as with any other word.

For instance:

```
a_buenas_horas a_buenas_horas RG A
a_causa_de a_causa_de SPS00 I
<accidente>_de_trabajo accidente_de_trabajo $1:NC I
<acabar>_de_VMN0000 acabar_de_$L3 $1:VMI I
Z_<vez> TIMES:$L1 Zu I
NC_SP_NC $L1_$L2_$L3 $1:NC I
```

The tag may be specified directly, or as a reference to the tag of some of the multiword components. In the previous example, the third multiword specification will build a multiword with any of the forms *accidente de trabajo* or *accidentes de trabajo*. The tag of the multiword will be that of its first form (`$1`) which starts with `NC` . This will assign the right singular/plural tag to the multiword, depending on whether the form was *accidente* or *accidentes*.

The lemma of the multiword may be specified directly, or as a reference to the form of lemma of some of the multiword components. In the previous example, the fourth multiword specification will build a multiword with phrases such as *acabo de comer*, *acababa de salir*, etc. The lemma will be `acabar_de_XXX` where `XXX` will be replaced with the lemma of the third multiword component (`$L3`).

The pattern can also depict PoS values for some components of the multiword. For instance, the second-to-last multiword above will detect any combination where the first word has PoS `Z` , and the second word has lemma *vez* (e.g *una vez*, *6 veces*, *cuarenta y dos veces*, etc.)

The last example pattern will capture any trigram where the first and third words are nouns (`NC`) and the middle word is a preposition (`SP`), e.g. *base de datos*, *compuesto de penicilina*, etc.

Note that patterns that specify lemmas or PoS will behave different depending on the value of the flag `OnlySelected` described above.

Lemma replacement strings can be `$F1` , `$F2` , `$F3` , etc. to select the lowercased form of any component, or `$L1` , `$L2` , `$L3` , etc. to select the lemma of any component. Component numbers can range from 1 to 9.

The last field states whether the multiword is ambiguous `A` or not `I` with respect to its segmentation (i.e. that it may be a multiword or not, depending on the context). The multiword is built in any case, but the ambiguity information is stored in the `word` object, so the calling applicacion can consult it and take the necessary decisions (e.g. un-glue the multiword) if needed.

Note that the described format allows only to encode multiwords whose components are adjacent. Non-adjacent multiwords are not supported.

Named Entity Recognition Module

There are two different modules able to perform NE recognition. They can be instantiated directly, or via a wrapper that will create the right module depending on the configuration file.

The API for the wrapper is the following:

```
class ner {
  public:
    /// Constructor
    ner(const std::wstring &cfgfile);
    /// Destructor
    ~ner();

    /// analyze given sentence
    void analyze(sentence &s) const;

    /// analyze given sentences
    void analyze(std::list<sentence> &ls) const;

    /// analyze sentence, return analyzed copy
    sentence analyze(const sentence &s) const;

    /// analyze sentences, return analyzed copy
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The parameter to the constructor is the absolute name of a configuration file, which must contain the desired module type (`basic` or `bio`) in a line enclosed by the tags `<Type>` and `</Type>` .

The rest of the file must contain the configuration options specific for the selected NER type, described below.

The `basic` module is simple and fast, and easy to adapt for use in new languages, provided capitalization is the basic clue for NE detection in the target language. The estimated performance of this module is about 85% correctly recognized named entities.

The `bio` module, is based on machine learning algorithms. It has a higher precision (over 90%), but it is remarkably slower than `basic`, and adaptation to new languages requires a training corpus plus some feature engineering.

Basic NER module (`np`)

The first NER module is the `np` class, which is a just an ATN that basically detects sequences of capitalized words, taking into account some functional words (e.g. *Bank of England*) and capitalization at sentence begginings.

It can be instantiated via the `ner` wrapper described above, or directly via its own API:

```
class np: public ner_module, public automat {
  public:
    /// Constructor, receives a configuration file.
    np(const std::string &cfgfile);

    /// Detect multiwords starting at given sentence position
    bool matching(sentence &s, sentence::iterator &p) const;

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The file that controls the behaviour of the simple NE recognizer consists of the following sections:

- Section `<FunctionWords>` lists the function words that can be embeeded inside a proper noun (e.g. preposisions and articles such as those in *Banco de España* or *Foundation for the Eradication of Poverty*). For instance:

```
<FunctionWords>
el
la
los
las
de
del
para
</FunctionWords>
```

- Section `<SpecialPunct>` lists the PoS tags (according to punctuation tags definition file) after which a capitalized word may be indicating just a sentence or clause beggining and not necessarily a named entity. Typical cases are colon, open parenthesis, dot, hyphen..

```
<SpecialPunct>
Fpa
Fp
Fd
Fg
</SpecialPunct>
```

- Section `<NE_Tag>` contains only one line with the PoS tag that will be assigned to the recognized entities. If the NE classifier is going to be used later, it will have to be informed of this tag at creation time.

```
<NE_Tag>
NP00000
</NE_Tag>
```

- Section `<Ignore>` contains a list of forms (lowercased) or PoS tags (uppercased) that are not to be considered a named entity even when they appear capitalized in the middle of a sentence. For instance, that can be used if the word *Spanish* in the sentence *He started studying Spanish two years ago* should not be considered named entity. If the words in the list appear with other capitalized words, they are considered to form a named entity (e.g. *An announcement of the Spanish Bank of Commerce was issued yesterday*). The same distinction applies to the word *I* in sentences such as: *whatever you say, I don't believe*, or *That was the death of Henry I*.

Each word or tag is followed by a 0 or 1 indicating whether the ignore condition is strict (0: non-strict, 1: strict). The entries marked as non-strict will have the behaviour described above. The entries marked as strict will never be considered named entities or NE parts.

For instance, the following `<Ignore>` section states that the word *I* is not to be a proper noun (e.g. *whatever you say, I don't believe*) unless some of its neighbour words are (e.g. *That was the death of Henry I*). It also states that any word with the RB tag, and any of the listed language names must never be considered as possible NEs.

```
<Ignore>
i  0
RB 1
english 1
dutch 1
spanish 1
</Ignore>
```

- Section `<Names>` contains a list of lemmas that may be names, even if they conflict with some of the heuristic criteria used by the NE recognizer. This is useful when they appear capitalized at sentence beggining. For instance, the basque name *Miren* (*Mary*) or the nickname *Pelé* may appear at the beggining of a Spanish sentence. Since both of them are verbal forms in Spanish, they would not be considered candidates to form named entities.

  Including the form in the `<Names>` section, causes the NE choice to be added to the possible tags of the form, giving the tagger the chance to decide whether it is actually a verb or a proper noun.

  ```
  <Names>
  miren
  pelé
  zapatero
  china
  </Names>
  ```

- Section `<Affixes>` contains a list of words that may be part of a NE -either prefixing or suffixing it- even if they are lowercased. For instance, this is the case of the word *don* in Spanish (e.g. *don Juan* should be a NE, even if *don* is lowercased), or the words *junior* or *jr.* in English (e.g. *Peter Grasswick jr.* should be a NE, even if *jr.* is lowercased).

The section should containt a word per line, followed by the keyword PRE or SUF stating whether the word may be attached before or after an NE. It a word should be either a prefix or a suffix, it must be declared in two different lines, one with each keyword.

```
<pre><Affixes>
don  PRE
doña PRE
jr.  SUF
<Affixes>
```

- Sections `<RE_NounAdj>` `<RE_Closed>` and `<RE_DateNumPunct>` allow to modify the default regular expressions for Part-of-Speech tags. This regular expressions are used by the NER to determine whether a sentence-beginning word has some tag that is Noun or Adj, or any tag that is a closed category, or one of date/punctuation/number. The default is to check against Eagles tags, thus, the recognizer will fail to identifiy these categories if your dictionary uses another tagset, unless you specify the right patterns to look for.

  For instance, if our dictionary uses Penn-Treebank-like tags, we should define:

  ```
  <pre><RE_NounAdj>
  ^(NN$|NNS|JJ)
  </RE_NounAdj>
  <RE_Closed>
  ^(D|IN|C)
  </RE_Closed>
  ```

- Section `<TitleLimit>` contains only one line with an integer value stating the length beyond which a sentence written *entirely* in uppercase will be considered a title and not a proper noun. Example:

  ```
  <TitleLimit>
  3
  </TitleLimit>
  ```

  If `TitleLimit=0` (the default) title detection is deactivated (i.e, all-uppercase sentences are always marked as named entities).

The idea of this heuristic is that newspaper titles are usually written in uppercase, and tend to have at least two or three words, while named entities written in this way tend to be acronyms (e.g. IBM, DARPA, ...) and usually have at most one or two words.

For instance, if `TitleLimit=3` the sequence *FREELING ENTERS NASDAC UNDER CLOSE OBSERVATION OF MARKET ANALYSTS* will not be recognized as a named entity, and will have its words analyzed independently. On the other hand, the sequence *IBM INC.*, having less than 3 words, will be considered a proper noun.

Obviously this heuristic is not 100% accurate, but in some cases (e.g. if you are analyzing newspapers) it may be preferrable to the default behaviour (which is not 100% accurate, either).

BIO NER module (`bioner`)

The machine-learning based NER module uses a classification algorithm to decide whether each word is at a NE begin (`B`), inside (`I`) or outside (`O`). Then, Viterbi algorithm is applied to guarantee sequence coherence.

It can be instantiated via the `ner` wrapper described above, or directly via its own API:

```
class bioner: public ner_module {
  public:
    /// Constructor, receives the name of the configuration
file.
    bioner ( const std::string & );

    /// analyze given sentence.
    void analyze(sentence &) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &)
const;
};
```

The configuration file sets the required model and lexicon files, which may be generated from a training corpus using the scripts provided with FreeLing.

Check file `src/utilities/train-nerc/README` and comments in the scripts to find out how to retrain the models and what is the meaning of each file.

The most important file in the set is the `.rgf` file, which contains a definition of the context features that must be extracted for each named entity. The feature rule language is described in the Feature Extractor section.

The sections of the configuration file for bioner module are:

- Section `<RGF>` contains one line with the path to the RGF file of the model. This file is the definition of the features that will be taken into account for NER.

  ```
  <RGF>
  ner.rgf
  </RGF>
  ```

- Section `<Classifier>` contains one line with the kind of classifier to use. Valid values are `AdaBoost` and `SVM`.

  ```
  <Classifier>
  Adaboost
  </Classifier>
  ```

- Section `<ModelFile>` contains one line with the path to the model file to be used. The model file must match the classifier type given in section `<Classifier>`.

  ```
  <ModelFile>
  ner.abm
  </ModelFile>
  ```

  The `.abm` files contain AdaBoost models based on shallow Decision Trees (see [CMP03] for details). You don't need to understand this, unless you want to enter into the code of the AdaBoost classifier.

  The `.svm` files contain Support Vector Machine models generated by `libsvm` [CL11]. You don't need to understand this, unless you want to enter into the code of `libsvm`.

- Section `<Lexicon>` contains one line with the path to the lexicon file of the learnt model. The lexicon is used to translate string-encoded features generated by the extractor to integer-encoded features needed by the classifier. The lexicon file is generated at training time.

```
<Lexicon>
ner.lex
</Lexicon>
```

  The `.lex` file is a dictionary that assigns a number to each symbolic feature used in the AdaBoost or SVM model. You don't need to understand this either unless you are a Machine Learning student or the like.

- Section `<UseSoftMax>` contains only one line with `yes` or `no`, indicating whether the classifier output must be converted to probabilities with the SoftMax function. Currently, AdaBoost models need that conversion, and SVM models do not.

```
<UseSoftMax>
yes
</UseSoftMax>
```

- Section `<Classes>` contains only one line with the classes of the model and its translation to `B`, `I`, `O` tag.

```
<Classes>
0 B 1 I 2 O
</Classes>
```

- Section `<NE_Tag>` contains only one line with the PoS tag that will be assigned to the recognized entities. If the NE classifier is going to be used later, it will have to be informed of this tag at creation time.

```
<NE_Tag>
NP00000
</NE_Tag>
```

- Section `<InitialProb>` Contains the probabilities of seeing each class at the begining of a sentence. These probabilities are necessary for the Viterbi algorithm used to annotate NEs in a sentence.

```
<InitialProb>
B 0.200072
I 0.0
O 0.799928
</InitialProb>
```

- Section `<TransitionProb>` Contains the transition probabilities for each class to each other class, used by the Viterbi algorithm.

```
<TransitionProb>
B B 0.00829346
B I 0.395481
B O 0.596225
I B 0.0053865
I I 0.479818
I O 0.514795
O B 0.0758838
O I 0.0
O O 0.924116
</TransitionProb>
```

- Section `<TitleLimit>` contains only one line with an integer value stating the length beyond which a sentence written *entirely* in uppercase will be considered a title and not a proper noun.

  See description of the same option for `basic` named entity recognizer above.

Quantity Recognition Module

The `quantities` class is an ATN that recognizes ratios, percentages, and physical or currency magnitudes (e.g. twenty per cent, 20%, one out of five, 1/5, one hundred miles per hour, etc.

This module depends on the numbers detection module. If numbers are not previously detected and annotated in the sentence, quantities will not be recognized.

This module, similarly to number recognition, is language dependent: That is, an ATN has to be programmed to match the patterns of ratio expressions in that language.

Currency and physical magnitudes can be recognized in any language, given the appropriate data file.

```
class quantities {
  public:
    /// Constructor: receives the language code, and the data
file.
    quantities(const std::string &lang, const std::string
&cfgfile);

    /// Detect magnitude expressions starting at given sentence
position
    bool matching(sentence &s, sentence::iterator &p) const;

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

Quantity Recognition Data File

This file contains the data necessary to perform currency amount and physical magnitude recognition. It consists of three sections: `<Currency>`, `<Measure>`, and `</MeasureNames>`.

Section `<Currency>` contains a single line indicating which is the code, among those used in section `<Measure>`, that stands for 'currency amount'. This is used to assign to currency ammounts a different PoS tag than physical magnitudes. E.g.:

```
<Currency>
CUR
</Currency>
```

Section `<Measure>` indicates the type of measure corresponding to each possible unit. Each line contains two fields: the measure code and the unit code. The codes may be anything, at user's choice, and will be used to build the lemma of the recognized quantity multiword.

E.g., the following section states that `USD` and `FRF` are of type `CUR` (currency), `mm` is of type `LN` (length), and `ft/s` is of type `SP` (speed):

```
<Measure>
CUR USD
CUR FRF
LN mm
SP ft/s
</Measure>
```

Finally, section `<MeasureNames>` describes which multiwords have to be interpreted as a measure, and which unit they represent. The unit must appear in section `<Measure>` with its associated code. Each line has the format: `multiword_description code tag` where:

- `multiword_description` is a multiword pattern as used in the multiwords detection file
- `code` is the type of magnitude the unit describes (currency, speed, etc.)
- `tag` is a constraint on the lemmatized components of the multiword, following the same conventions than in multiwords detection file.

E.g.,

```
<MeasureNames>
french_<franc> FRF $2:N
<franc> FRF $1:N
<dollar> USD $1:N
american_<dollar> USD $2:N
us_<dollar> USD $2:N
<milimeter> mm $1:N
<foot>_per_second ft/s $1:N
<foot>_Fh_second ft/s $1:N
<foot>_Fh_s ft/s $1:N
<foot>_second ft/s $1:N
</MeasureNames>
```

This section will produce results such as the following:

```
234_french_francs CUR_FRF:234 Zm
one_dollar CUR_USD:1 Zm
two_hundred_fifty_feet_per_second SP_ft/s:250 Zu
```

Quantity multiwords will be recognized only when following a number, that is, in the sentence *They got a lot of french francs*, the multiword `french_francs` won't be recognized since the sentence is specifying determined quantity of that measure unit.

It is important to note that the lemmatized multiword expressions (the ones that containt angle brackets) will only be recognized if the lemma is present in the dictionary with its corresponding inflected forms.

Probability Assignment and Unknown Word Guesser Module

This class ends the morphological analysis subchain, and has two functions: first, it assigns an a priori probability to each analysis of each word. These probablities will be needed for the PoS tagger later. Second, if a word has no analysis (none of the previously applied modules succeeded to analyze it), this module tries to guess which are its possible PoS tags, based on the word ending.

```
class probabilities {
  public:
    /// Constructor: receives the name of the file
    // containing probabilities, and a threshold.
    probabilities(const std::string &cfgfile, double thresh);

    /// Assign probabilities for each analysis of given word
    void annotate_word(word &w) const;

    /// Turn guesser on/of
    void set_activate_guesser(bool b);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The method `set_activate_guesser` will turn on/off the guessing of likely PoS tags for words with no analysis. Note that the guesser is turned on/off for any thread using the same probabilities instance.

The constructor receives:

- The probabilities file name: The file that contains all needed configuration and statistical information. This file can be generated from a tagged training corpus using the scripts in `src/utilities/train-tagger`. Its format is described below.
- A threshold: This is used for unknown words, when the probability of each possible tag has been estimated by the guesser according to word endings, tags with a value lower than this threshold are discarded.

Lexical Probabilities File

This file can be generated from a tagged corpus using the training script provided in FreeLing package, located at `src/utilities/train-tagger/bin/TRAIN.sh`.

See `src/utilities/train-tagger/README` to find out how to use it.

The probabilities file has the following sections:

- Optional section `<LemmaPreferences>` and contains a list of pairs of lemmas. The meaning of each pair is that the first element is prefereble to the second in case the tagger can not decide between them and is forced to.

  For instance, the section:

  ```
  <LemmaPreferences>
  salir salgar
  </LemmaPreferences>
  ```

  solves the ambiguity for Spanish word salgo, which may correspond to indicative first person singular of verb salir (go out), or to exactly the same tense of verb salgar (feed salt to cattle). Since the PoS tag is the same for both lemmas, the tagger can not decide which is the right one. This preference solves the dilemma in favour of salir (go out), which is more frequent.

- Optional section `<PoSPreferences>` and contains a list of pairs of PoS tags. The meaning of each pair is that the first element is prefereble to the second in case the tagger can not decide between them and is forced to. The POS preference is only used when the tie can not be solved using lemma preferences.

For instance, the section

```
<PosPreferences>
VMII3S0 VMII1S0
</PosPreferences>
```

helps solving cases as the past tense for Spanish verbs such as cantaba (I/he sung), which are shared by first and third person. In this case, if the tagger is not able to make a decision a preference is set for 3rd person (which is more frequent in standard text).

- Section `<TagsetFile>` : This section contains a single line with the path to a tagset description file to be used when computing short versions for PoS tags. If the path is relative, the location of the lexical probabilities file is used as the base directory.
- Section `<FormTagFreq>` : Probability data of some high frequency forms.

  If the word is found in this list, lexical probabilities are computed using data in `<FormTagFreq>` section.

  The list consists of one form per line, each line with format:
  `form ambiguity-class, tag1 #observ1 tag2 #observ2 ...`
  E.g.:
  `japonesas AQ-NC AQ 1 NC 0`

  Form probabilities are smoothed to avoid zero-probabilities.
- Section `<ClassTagFreq>` : Probability data of ambiguity classes.

  If the word is not found in the `<FormTagFreq>` , frequencies for its ambiguity class are used.

  The list consists of class per line, each line with format:
  `class tag1 #observ1 tag2 #observ2 ...`
  E.g.:
  `AQ-NC AQ 2361 NC 2077`

  Class probabilities are smoothed to avoid zero-probabilities.
- Section `<SingleTagFreq>` : Unigram probabilities.

  If the ambiguity class is not found in the `<ClassTagFreq>` , individual frequencies for its possible tags are used.

  One tag per line, each line with format:
  `tag #observ`
  E.g.:
  `AQ 7462`

Tag probabilities are smoothed to avoid zero-probabilities.

- Section `<Theeta>` : Value for parameter theeta used in smoothing of tag probabilities based on word suffixes.

  If the word is not found in dictionary (and so the list of its possible tags is unknown), the distribution is computed using the data in the `<Theeta>` , `<Suffixes>` , and `<UnknownTags>` sections.

  The section has exactly one line, with one real number.
  E.g.

```
<Theeta>
0.00834
</Theeta>
```

- Section `<BiassSuffixes>` : Weighted interpolation factor between class probability and word suffixes.

  The section has exactly one line, with one real number.
  E.g.

```
<BiassSuffixes>
0.4
</BiassSuffixes>
```

  This section is optional. If ommited, used default value is 0.3.

  The probability of the tags belonging to words unobserved in the training corpus, is computed backing off to the distribution of all words with the same ambiguity class. This obviously overgeneralizes and for some words, the estimated probabilities may be rather far from reality.

  To palliate this overgeneralization, the ambiguity class probabilities can me interpolated with the probabilities assigned by the guesser according to the word suffix.

  This parameter specifies the weight that suffix information is given in the iterpolation, i.e. if `BiassSuffixes=0` only the ambiguity class information is used. If `BiassSuffixes=1` , only the probabilities provided by the guesser are used.

- Section `<Suffixes>` : List of suffixes obtained from a train corpus, with information about which tags were assigned to the word with that suffix.

The list has one suffix per line, each line with format:
`suffix #observ tag1 #observ1 tag2 #observ2 ...`
E.g.
`orada 133 AQ0FSP 17 VMP00SF 8 NCFS000 108`

- Section `<UnknownTags>` : List of open-category tags to consider as possible candidates for any unknown word.

  One tag per line, each line with format:
  `tag #observ`
  The tag is the complete label. The count is the number of occurrences in a training corpus.
  E.g.
  `NCMS000 33438`

- Section `<LidstoneLambdaLexical>` : specifies the λ parameter for Lidstone's Law smoothing of seen words

  The section has exactly one line, with one real number.
  E.g.

```
<LidstoneLambdaLexical>
0.2
</LidstoneLambdaLexical>
```

  This section is optional. If ommited, used default value is 0.1.

  This parameter is used only to smooth the lexical probabilities of words that have appeared in the training corpus, and thus are listed in the `<FormTagFreq>` section described above.

- Section `<LidstoneLambdaClass>` specifies the λ parameter for Lidstone's Law smoothing of unseen words

  The section has exactly one line, with one real number. E.g.

```
<LidstoneLambdaLexical>
0.5
</LidstoneLambdaLexical>
```

  This section is optional. If ommited, used default value is 0.1.

This parameter is used only to smooth the lexical probabilities of words that have not appeared in the training corpus, and thus are not listed in the `<FormTagFreq>` section. In these cases, information from appropriate ambiguity class in `<ClassTagFreq>` section is used.

Alternative Suggestion Module

This module is able to retrieve from its dictionary the entries most similar to the input form. The similarity is computed according to a configurable string edit distance (SED) measure.

The alternatives module can be created to perform a direct search of the form in a dictionary, or either to perform a search of the phonetic transcription of the form in a dictionary of phonetic transcriptions. In the later case, the orthographic forms corresponding to the phonetically similar words are returned. For instance, if a mispelled word such as spid is found, this module will find out that it sounds very close to some correct words in the dictionary (e.g. speed, spit), and return the correctly spelled alternatives. This module is based on the fast search algorithms on FSMs included in the finite-state libray FOMA.

The API for this module is the following:

```
class alternatives {
  public:
    /// Constructor
    alternatives(const std::wstring &cfgfile);

    /// Destructor
    ~alternatives();

    /// direct access to results of underlying FSM to retrieve
    /// most similar words to the given one
    void get_similar_words(const std::wstring &form,
                           std::list<std::pair<std::wstring,int>
> &results) const;

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

This module will find alternatives for words in the sentences, and enrich them with a list of forms, each with the corresponding SED value. The forms are added to the `alternatives` member of class `word`, which is a `std::list<pair<std::wstring,int>>`. The list can be traversed using the iterators `word::alternatives_begin()` and `word::alternatives_end()`.

The constructor of this module expects a configuration file containing the following sections:

- Section `<General>` contains values for general parameters, expressed in lines of the form: `key value`.

  More specifically, it must contain the following pairs `key value`:

- `Type (orthographic|phonetic)`
  Defines whether the similar words must be searched using direct SED between of orthographic forms, or between their phonetic encoding.

- `Dictionary filename`
  Dictionary where the candidate alternatives words are going to be searched. Only needed if `Type` is `orthographic`.

  The dictionary can be any file containing one form per line. Only first field in the line will be considered, which makes it possible to use a basic FreeLing dictionary file since the morphological information will be ignored.

- `PhoneticDictionary filename`
  Dictionary where the candidate alternatives words are going to be searched. Only needed if `Type` is `phonetic`.

- `PhoneticRule filename`
  Configuration file for a phonetic encoding module that will be used to encode the input forms before the search. Only needed if `Type` is `phonetic`.

  The `PhoneticDictionary` must contain one phonetic form per line, followed by a list of orthographic forms mapping to that sound. E.g., valid lines are:

  ```
  f@Ur fore four
  tu too two
  ```

- Section `<Distance>` contains `key value` lines stating parameters related to the SED measure to use:

  - `CostMatrix filename`
    File where the SED cost matrix to be used. The `CostMatrix` file must comply with FOMA requirements for cost matrices (see FOMA documentation, or examples provided in `data/common/alternatives` in FreeLing tarball).

  - `Threshold integer`
    Value for the maximum distance of desired alternatives. Note that a very high value will cause the module to produce a long list of similar words, and a too low value may result in no similar forms found.

  - `MaxSizeDiff integer`
    Similar strings with a length difference greater than this parameter will be filtered out of the result. To deactivate this feature, just set the value to a large number (e.g. 99).

- Section `<Target>` contains `key value` lines describing which words in the sentence must be checked for similar forms.
    - `UnknownWords (yes|no)`
      states whether similar forms are to be searched for unknown words (i.e. words that didn't receive any analysis from any module so far).
    - `KnownWords regular-expression`
      states which words with analysis have to be checked. The regular expression is matched against the PoS tag of the words. If the regular-expression is `none`, no known word is checked for similar forms.

Sense Labelling Module

This module searches the lemma of each analysis in a sense dictionary, and enriches the analysis with the list of senses found there.

Note that this is not disambiguation, all senses for the lemma are returned.

The module receives a file containing several configuration options, which specify the sense dictionary to be used, and some mapping rules that can be used to adapt FreeLing PoS tags to those used in the sense dictionary.

FreeLing provides WordNet-based [Fel98,Vos98] dictionaries, but the results of this module can be changed to any other sense catalogue simply providing a different sense dictionary file.

```
class senses {
  public:
    /// Constructor: receives the name of the configuration file
    senses(const std::string &cfgfile);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor of this class receives the name of a configuration file which is expected to contain the following sections:

- A section `<WNposMap>` with the mapping rules of FreeLing PoS tags to sense dictionary PoS tags.
  The format of the mapping rules is described in section Semantic Database.

- A section `<DataFiles>` containing the following pairs `keyword value` :
  - `SenseDictFile filename`
    Sense dictionary to use. E.g.

    ```
    <DataFiles>
    SenseDictFile  ./senses30.src
    </DataFiles>
    ```

    The format of the sense dictionary is described in section Semantic Database.
  - `formDictFile filename`
    Form dictionary to use if mapping rules in `WNposMap` require the use of a form dictionary.
- A section `<DuplicateAnalysis>` containing a single line with either `yes` or `no` , stating whether the analysis with more than one senses must be duplicated. If this section is ommitted, `no` is used as default value. The effect of activating this option is described in the following example:

  For instance, the word crane has the follwing analysis:

  ```
  crane NN  0.833
  crane VB  0.083
  crane VBP 0.083
  ```

  If the list of senses is simply added to each of them (that is, `DuplicateAnalysis` is set to `false`), you will get:

  ```
  crane NN  0.833  02516101:01524724
  crane VB  0.083  00019686
  crane VBP 0.083  00019686
  ```

  But if you set `DuplicateAnalysis` to true, the `NN` analysis will be duplicated for each of its possible senses:

  ```
  crane NN  0.416  02516101
  crane NN  0.416  01524724
  crane VB  0.083  00019686
  crane VBP 0.083  00019686
  ```

Word Sense Disambiguation Module

This module performs word-sense-disambiguation on content words in given sentences. This module is to be used if word sense disambiguation (WSD) is desired. If no disambiguation (or basic most-frequent-sense disambiguation) is needed, the senses module is a lighter and faster option.

The module is an implementation of UKB algorithm [AS09]. UKB relies on a semantic relation network (in this case, WN and XWN) to disambiguate the most likely senses for words in a text using PageRank algorithm. See [AS09] for details on the algorithm.

The module expects the input words to have been annotated with a list of candidate senses by the senses module, and will rank the candidate senses according to the PageRank for each sense. The rank value is also provided in the result. The top-ranked sense(s) can be selected as the most likely disambiguation for the word in that context

The API of the class is the following:

```
class ukb {
  public:
    /// Constructor. Receives a configuration file for UKB
    ukb(const std::string &cfgfile);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor receives a file name where module configuration options are found. The contents of the configuration files are the following:

- A section `<PageRankParameters>` specifying values for UKB stopping criteria. E.g.:

```
<PageRankParameters>
Epsilon 0.03
MaxIterations 10
Damping 0.85
</PageRankParameters>
```

The `Epsilon` value controls the precision with with the end of PageRank iterations is decided.
`MaxIterations` controls the maximum number of PageRank iterations, even is no convergence is reached.
The `Damping` parameter is the standard parameter in PageRank algorithm.

- A section `<RelationFile>` specifying the knowledge base required by the algorithm. This section must contain a single line with the path to a file containing a list of relations between senses.

```
<RelationFile>
../common/xwn.dat
</RelationFile>
```

The path may be absolute, or relative to the position of the ukb module configuration file.

Given file must contain the semantic relationship graph to load. It is a text filecontaining pairs of related senses (WN synsets in this case). Relations are neither labelled nor directed.

An example of the content of this file is:

```
00003431-v 14877585-n
00003483-r 00104099-r
00003483-r 00890351-a
```

Part-of-Speech Tagger Module

There are two different modules able to perform PoS tagging. The application should decide which method is to be used, and instantiate the right class.

The first PoS tagger is the `hmm_tagger` class, which is a classical trigam Markovian tagger, following [Bra00].

The second module, named `relax_tagger`, is a hybrid system capable to integrate statistical and hand-coded knowledge, following [Pad98].

The `hmm_tagger` module is somewhat faster than `relax_tagger`, but the later allows you to add manual constraints to the model. Its API is the following:

Both classes are derived from the `POS_tagger` abstract class:

```
class POS_tagger {
  public:
    POS_tagger(bool,unsigned int);
    virtual ~POS_tagger() {};

    /// Do actual disambiguation
    virtual void annotate(sentence &) const =0;

    /// analyze given sentence.
    virtual void analyze(sentence &s) const;

    /// analyze given sentences.
    virtual void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    virtual sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    virtual std::list<sentence> analyze(const
std::list<sentence> &ls) const;
};
```

Other PoS methods can be added deriving new classes from `POS_tagger`.

Hidden Markov Model PoS Tagger

The `hmm_tagger` implements a classical trigam Markovian tagger.

Its API is:

```
class hmm_tagger: public POS_tagger {
  public:
    /// Constructor
    hmm_tagger(const std::string &hmmfile,
               bool retok,
               unsigned int force,
               unsigned int kb=1);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;

    /// given an analyzed sentence find out probability
    /// of the k-th best sequence
    double SequenceProb_log(const sentence &s, int k=0) const;
};
```

The `hmm_tagger` constructor receives the following parameters:

- The HMM file, which containts the model parameters. The format of the file is described below. This file can be generated from a tagged corpus using the script `src/utilities/train-tagger/bin/TRAIN.sh` provided in FreeLing package. See `src/utilities/train-tagger/README` to find out the details.
- A boolean stating whether words that carry retokenization information (e.g. set by the dictionary or affix handling modules) must be retokenized (that is, splitted in two or more

words) after the tagging.

- An integer stating whether and when the tagger must select only one analysis in case of ambiguity. Possbile values are: `FORCE_NONE` (or 0): no selection forced, words ambiguous after the tagger, remain ambiguous. `FORCE_TAGGER` (or 1): force selection immediately after tagging, and before retokenization. `FORCE_RETOK` (or 2): force selection after retokenization.
- An integer stating how many best tag sequences the tagger must try to compute. If not specified, this parameter defaults to 1. Since a sentence may have less possible tag sequences than the given *k* value, the results may contain a number of sequences smaller than *k*.

HMM-Tagger Parameter File

This file contains the statistical data for the Hidden Markov Model, plus some additional data to smooth the missing values. Initial probabilities, transition probabilities, lexical probabilities, etc.

The file may be generated by your own means, or using a tagged corpus and the training script provided in FreeLing package: `src/utilities/train-tagger/bin/TRAIN.sh`. See `src/utilities/train-tagger/README` for details.

The file has eight sections which contain -among other things- the paremeters of the HMM (e.g. the tag (unigram), bigram, and trigram probabilities used in Linear Interpolation smoothing by the tagger to compute state transition probabilities (α_{ij} parameters of the HMM):

- Section `<TagsetFile>`. This section contains a single line with the path to a tagset description file to be used when computing short versions for PoS tags. If the path is relative, the location of the lexical probabilities file is used as the base directory.
  This section has to appear before section `<Forbidden>`.

- Section `<Tag>`. List of unigram tag probabilities (estimated via your preferred method). Each line is a tag probability *P(t)* with format
  `Tag Probability`
  Lines for tags `0` (sentence beginning) and `x` (unobserved tags) must be included. E.g.

  ```
  0 0.03747
  AQ 0.00227
  NC 0.18894
  x 1.07312e-06
  ```

- Section `<Bigram>`. List of bigram transition probabilities (estimated via your preferred method). Each line is a transition probability, with the format:
  `Tag1.Tag2 Probability` Tag `0` (zero) indicates sentence-begining.

  E.g.:

  - The following line indicates the transition probability between a sentence start and the tag of the first word being `AQ`.
    `0.AQ 0.01403`
  - The following line indicates the transition probability between two consecutive tags.
    `AQ.NC 0.16963`
- Section `<Trigram>`. List of trigram transition probabilities (estimated via your preferred method). Each line is a transition probability, with the format:
  `Tag1.Tag2.Tag3 Probability`
  Tag `0` (zero) indicates sentence-begining.

  E.g.:

  - The following line indicates the probability that a word has `NC` tag just after a `0.AQ` sequence.
    `0.AQ.NC 0.204081`
  - The following line indicates the probability of a tag `SP` appearing after two words tagged `DA` and `NC`.
    `DA.NC.SP 0.33312`
- Section `<Initial>`. List of initial state probabilities (estimated via your preferred method), i.e. the π_i parameters of the HMM. Each line is an initial probability, with the format:
  `InitialState LogProbability`

  Each `InitialState` is a PoS-bigram code with the form `0.tag`. Probabilities are given in logarithmic form to avoid underflows.

  E.g.:

  - The following line indicates the probability that the sequence starts with a determiner.
    `0.DA -1.744857`
  - The following line indicates the probability that the sequence starts with an unknown tag.
    `0.x -10.462703`
- Section `<Word>`. Contains a list of word probabilities *P(w)* (estimated via your preferred method). It is used, toghether with the tag probabilities above, to compute emission probabilities (b_{iw} parameters of the HMM).

Each line is a word probability *P(w)* with format: `word LogProbability` .
A special line for `<UNOBSERVED_WORD>` must be included.
Sample lines for this section are:

```
afortunado -13.69500
sutil -13.57721
<UNOBSERVED_WORD> -13.82853
```

- Section `<Smoothing>` contains three lines with the coefficients used for linear interpolation of unigram (`c1`), bigram (`c2`), and trigram (`c3`) probabilities. The section looks like:

```
<Smoothing>
c1 0.120970620869314
c2 0.364310868831106
c3 0.51471851029958
</Smoothing>
```

- Section `<Forbidden>` is the only that is not generated by the training scripts, and is supposed to be manually added (if needed). The utility is to prevent smoothing of some combinations that are known to have zero probability.

  Lines in this section are trigrams, in the same format than above: `Tag1.Tag2.Tag3`

  Trigrams listed in this section will be assigned zero probability, and no smoothing will be performed. This will cause the tagger to avoid any solution including these subsequences.

  The first tag may be a wildcard (`*`), which will match any tag, or the tag `0` which denotes sentence beginning. These two special tags can only be used in the first position of the trigram.

  In the case of an EAGLES tagset, the tags in the trigram may be either the short or the long version. The tags in the trigram (except the special tags `*` and `0`) can be restricted to a certain lemma, suffixing them with the lemma in angle brackets.

  For instance, the following rules will assign zero probability to any sequence containing the specified trigram:

  - `*.PT.NC` : a noun after an interrogative pronoun.
  - `0.DT.VMI` : a verb in indicative following a determiner just after sentence beggining.
  - `SP.PP.NC` : a noun following a preposition and a personal pronoun.

Similarly, the set of rules:

```
*.VAI<haber>.NC
*.VAI<haber>.AQ
*.VAI<haber>.VMP00SF
*.VAI<haber>.VMP00PF
*.VAI<haber>.VMP00PM
```

will assign zero probability to any sequence containing the verb *haber* (*to have*) tagged as an auxiliar (`VAI`) followed by any of the listed tags. Note that the masculine singular participle is not excluded, since it is the only allowed after an auxiliary *haber*. This will force the tagger to pick the `VM` tag for *haber* when it is not followed by a masculine singular participle.

Relaxation Labelling PoS Tagger

The `relax_tagger` module can be tuned with hand written constraint, but it has under half the speed of `hmm_tagger` . It is not able to produce *k* best sequences. The advantadges it offers is the ability to biass/correct the model adding linguistically mootivated manual constraints.

```
class relax_tagger : public POS_tagger {
  public:
    /// Constructor, given the constraint file and config
parameters
    relax_tagger(const std::string &cfgfile,
                 int m,
                 double f,
                 double r,
                 bool retok,
                 unsigned int force);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The `relax_tagger` constructor receives the following parameters:

- The constraint file. The format of the file is described below. This file can be generated from a tagged corpus using the script `src/utilities/train-tagger/bin/TRAIN.sh` provided in FreeLing package. See `src/utilities/train-tagger/README` for details.
- An integer `m` stating the maximum number of iterations to wait for convergence before stopping the disambiguation algorithm.
- A real number `f` representing the scale factor of the constraint weights.
- A real number `r` representing the threshold under which any changes will be considered too small. Used to detect convergence.
- A boolean `retok` stating whether words that carry retokenization information (e.g. set by the dictionary or affix handling modules) must be retokenized (that is, splitted in two or more words) after the tagging.
- An integer stating whether and when the tagger must select only one analysis in case of ambiguity. Possbile values are: `FORCE_NONE` (or 0): no selection forced, words

ambiguous after the tagger, remain ambiguous. `FORCE_TAGGER` (or 1): force selection immediately after tagging, and before retokenization. `FORCE_RETOK` (or 2): force selection after retokenization.

The iteration number, scale factor, and threshold parameters are very specific of the relaxation labelling algorithm. Refer to [Pad98] for details.

Relaxation-Labelling Constraint Grammar File

The syntax of the file is based on that of Constraint Grammars [KVHA95], but simplified in many aspects, and modified to include weighted constraints.

An initial file based on statistical constraints may be generated from a tagged corpus using the `src/utilities/train-tagger/bin/TRAIN.sh` script provided with FreeLing. Later, hand written constraints can be added to the file to improve the tagger behaviour.

The file consists of two sections: `SETS` and `CONSTRAINTS`.

Set definition

The `SETS` section consists of a list of set definitions, each of the form:
`Set-name = element1 element2 ... elementN;`

Where the `Set-name` is any alphanumeric string starting with a capital letter, and the elements are either forms, lemmas, plain PoS tags, or senses. Forms are enclosed in parenthesis -e.g. `(comimos)`-, lemmas in angle brackets -e.g. `<comer>`-, PoS tags are alphanumeric strings starting with a capital letter -e.g. `NCMS000`-, and senses are enclosed in square brackets -e.g. `[00794578]`. The sets must be homogeneous: That is, all the elements of a set have to be of the same kind.

Examples of set definitions:

```
DetMasc = DA0MS0 DA0MP0 DD0MS0 DD0MP0 DI0MS0 DI0MP0 DP1MSP
DP1MPP
          DP2MSP DP2MPP DT0MS0 DT0MP0 DE0MS0 DE0MP0 AQ0MS0
AQ0MP0;
VerbPron = <dar_cuenta> <atrever> <arrepentir> <equivocar>
<inmutar>
           <morir> <ir> <manifestar> <precipitar> <referir>
<venir>;
Animal = [00008019] [00862484] [00862617] [00862750] [00862871]
[00863425]
         [00863992] [00864099] [00864394] [00865075] [00865379]
[00865569]
         [00865638] [00867302] [00867448] [00867773] [00867864]
[00868028]
         [00868297] [00868486] [00868585] [00868729] [00911889]
[00985200]
         [00990770] [01420347] [01586897] [01661105] [01661246]
[01664986]
         [01813568] [01883430] [01947400] [07400072] [07501137];
```

Constraint definition

The `CONSTRAINTS` section consists of a series of context constraits, each of the form: `weight core context;`

Where:

- `weight` is a real value stating the compatibility (or incompatibility if negative) degree of the core `label` with the `context`.
- `core` indicates the analysis or analyses in a word that will be affected by the constraint. It may be:
  - Plain tag: A plain complete PoS tag, e.g. `VMIP3S0`
  - Wildcarded tag: A PoS tag prefix, right-wilcarded, e.g. `VMI*`, `VMIP*`.
  - Lemma: A lemma enclosed in angle brackets, optionaly preceded by a tag or a wildcarded tag. e.g. `<comer>`, `VMIP3S0<comer>`, `VMI*<comer>` will match any word analysis with those tag/prefix and lemma.
  - Form: Form enclosed in parenthesis, preceded by a PoS tag (or a wilcarded tag). e.g. `VMIP3S0(comió)`, `VMI*(comió)` will match any word analysis with those tag/prefix and form. Note that the form alone is not allowed in the rule core, since the rule would not distinguish among different analysis of the same form.

    - Sense: A sense code enclosed in square brackets, optionaly preceded by a tag or a wildcarded tag. e.g. `[00862617]`, `NCMS000[00862617]`, `NC*[00862617]` will match any word analysis with those tag/prefix and sense.
- `context` is a list of conditions that the context of the word must satisfy for the constraint to be applied. Each condition is enclosed in parenthesis and the list (and thus the constraint) is finished with a semicolon. Each condition has the form:
  `(position terms)`
  or either:
  `(position terms barrier terms)` `(position terms barrier terms)`
  Conditions may be negated using the token not, i.e.
  `(not pos terms)`
  `(not position terms barrier terms)`

  Where:

    - `position` is the relative position where the condition must be satisfied: `-1` indicates the previous word and `+1` the next word. A position with a star (e.g. `-2*`) indicates that any word is allowed to match starting from the indicated position and advancing towards the beggining/end of the sentence.
    - `terms` is a list of one or more terms separated by the token `or`. Each term may be:
        - Plain tag: A plain complete PoS tag, e.g. `VMIP3S0`
        - Wildcarded tag: A PoS tag prefix, right-wilcarded, e.g. `VMI*`, `VMIP*`.
        - Lemma: A lemma enclosed in angle brackets, optionaly preceded by a tag or a wildcarded tag. e.g. `<comer>`, `VMIP3S0<comer>`, `VMI*<comer>` will match any word analysis with those tag/prefix and lemma.
        - Form: Form enclosed in parenthesis, optionally preceded by a PoS tag (or a wilcarded tag). e.g. `(comió)`, `VMIP3S0(comió)`, `VMI*` will match any word analysis with those tag/prefix and form. Note that -contrarily to when defining the rule core- the form alone is allowed in the context.
        - Sense: A sense code enclosed in square brackets, optionaly preceded by a tag or a wildcarded tag. e.g. `[00862617]`, `NCMS000[00862617]`, `NC*[00862617]` will match any word analysis with those tag/prefix and sense.
        - Set reference: A name of a previously defined SET in curly brackets. e.g. `{DetMasc}`, `{VerbPron}` will match any word analysis with a tag, lemma or sense in the specified set.
    - `barrier` inhibits the application of the rule if a match for the associated term conditions is found between the focus word and the word matching the rule conditions.

Note that the use of sense information in the rules of the constraint grammar (either in the core or in the context) only makes sense when this information distinguishes one analysis from another. If the sense tagging has been performed with the option `DuplicateAnalysis=no`, each PoS tag will have a list with all analysis, so the sense information will not distinguish one analysis from the other (there will be only one analysis with that sense, which will have at the same time all the other senses as well). If the option `DuplicateAnalysis` was active, the sense tagger duplicates the analysis, creating a new entry for each sense. So, when a rule selects an analysis having a certain sense, it is unselecting the other copies of the same analysis with different senses.

Examples

The next constraint states a high incompatibility for a word being a definite determiner (`DA*`) if the next word is a personal form of a verb (`VMI*`):

```
-8.143 DA*
       (1 VMI*);
```

The next constraint states a very high compatibility for the word *mucho* (*much*) being an indefinite determiner (`DI*`) -and thus not being a pronoun or an adverb, or any other analysis it may have- if the following word is a noun (`NC*`):

```
60.0 DI*(mucho)
     (1 NC*);
```

The next constraint states a positive compatibility value for a word being a noun (`NC*`) if somewhere to its left there is a determiner or an adjective (`DA* or AQ*`), and between them there is not any other noun:

```
5.0 NC*
    (-1* DA* or AQ* barrier NC*);
```

The next constraint states a positive compatibility value for a word being a masculine noun (`NCM*`) if the word to its left is a masculine determiner. It refers to a previously defined SET which should contain the list of all tags that are masculine determiners. This rule could be useful to correctly tag Spanish words which have two different NC analysis differing in gender: e.g. *el cura* (*the priest*) vs. *la cura* (*the cure*):

```
5.0 NCM*
    (-1* DetMasc);
```

The next constraint adds some positive compatibility to a 3rd person personal pronoun being of undefined gender and number (`PP3CNA00`) if it has the possibility of being masculine singular (`PP3MSA00`), the next word may have lemma *estar* (*to be*), and the second word to the right is not a gerund (`VMG*`). This rule is intended to solve the different behaviour of the Spanish clitic pronoun *lo* (it) in sentences such as *¿Cansado? Si, lo estoy.* (*Tired? Yes, I am [it]*) or *lo estoy viendo.* (*I am watching it*).

```
0.5 PP3CNA00
    (0 PP3MSA00)
    (1 <estar>)
    (not 2 VMG*);
```

Phonetic Encoding Module

The phonetic encoding module enriches words with their SAMPA phonetic codification.

The module applies a set of rules to transform the written form to the output phonetic form, thus the rules can be changed to get the output in any other phonetic alphabet. The module can also use an exception dictionary to handle forms that do not follow the default rules, or for languages with highly irregular ortography.

The API of the module is the following:

```
class phonetics {

 public:
  /// Constructor, given config file
  phonetics(const std::wstring &cfgfile);

  /// Returns the phonetic sound of the word
  std::wstring get_sound(const std::wstring &form) const;

  /// analyze given sentence, enriching words with phonetic
encoding
  void analyze(sentence &s) const;

  /// analyze given sentences
  void analyze(std::list<sentence> &ls) const;

  /// analyze sentence, return analyzed copy
  sentence analyze(const sentence &s) const;

  /// analyze sentences, return analyzed copy
  std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor receives a configuration file that contains the transformation rules to apply and the exception dictionary.

The module can be used to transform a single string (method `get_sound`) or to enrich all words in a sentence (or sentence list) with their phonetic information.

The phonetic encoding is stored in the `word` object and can be retrieved using the method `get_ph_form` in the `word` class.

Phonetic encoder configuration file:

The configuration file contains two kinds of sections:

- Section `<Exceptions>` contains an exception dictionary. This section is optional, and if it occurs, it must be just once.
  Each entry in the exception dictionary contains two fields: a lowercase word form and the output phonetic encoding.
  E.g.:

  ```
  addition @dISIn
  varieties B@raIItiz
  worcester wUst@r
  ```

  If a word form is found in the exceptions dictionary, the corresponding phonetic string is returned and no transformation rules are applied.

- Section `<Rules>` contain transformation rules that convert from orthogaphic to phonetic writting.
  The file may contain one or more rulesets delimited by `<Rules>` and `</Rules>`.
  Rulesets are applied in the order they are defined. Each ruleset is applied on the result of the previous.

  Rulesets can contain two kind of lines: Category definitions and rules.

  - Category definitions are of the form `X=abcde` and define a set of characters under a single name. Category names must have exactly one character, which should not be part of the input or output alphabet to avoid ambiguities. e.g.:

    ```
    U=aeiou
    V=aeiouäëïöüâêîôûùò@
    L=äëïöüäëïöüäëïöüùò@
    S=âêîôûâêîôûâêîôûùò@
    A=aâä
    E=eêë
    ```

Categories are only active for rules in the same ruleset where the category is defined.

- Rules have the form: `source/target/context`.
  Both `source` and `target` may be either a category name or a terminal string.

  Simple string rules replace the `source` string with the `target` string if and only if it occurs in the given `context`.

  Contexts must contain a `_` symbol indicating where the source string is located. They may also contain characters, categories, and the symbols `^` (word beginning), or `$` (word end). The empty context `_` is always satisfied.

  Rules can only change terminal strings to terminal strings, or categories to categories (i.e. both `source` and `target` have to be of the same type). If a category is to be changed to another category, they should contain the same number of characters. Otherwise the second category will have its last letter repeated until it has the same length as the first (if it is shorter), or characters in the second category that don't match characters in the first will be ignored.

  Some example rules for English:

  ```
  qu/kw/_
  h//^r_
  a/ò/_lmV$
  U/L/C_CV
  ```

  The first rule `qu/kw/_` replaces string `qu` with `kw` in any context.

  The second rule `h//^r_` removes character `h` when it is preceeded by `r` at word beginning.

  Rule `a/ò/_lmV$` replaces `a` with `ò` when followed by `l`, `m`, or any character in category `V` at the end of the word.

  Rule `U/L/C_CV` replaces any character in category `U` with the character in the same position in category `L`, when preceeded by any character in category `C` and followed by any character in category `C` plus any character in category `V`.

  Note that uppercase characters for categories is just a convention. An uppercase letter may be a terminal symbol, and a lowercase may be a category name. Non-alphabetical characters are also allowed. If a character is not defined as a category name, it will be considered a terminal character.

Named Entity Classification Module

The mission of the Named Entity Classification module is to assing a class to named entities in the text. It is a Machine-Learning based module, so the classes can be anything the model has been trained to recognize.

When classified, the PoS tag of the word is changed to the label defined in the model.

This module depends on a NER module being applied previously. If no entities are recognized, none can be classified.

Models provided with FreeLing distinguish four classes: Person (tag `NP00SP0`), Geographical location (`NP00G00`), Organization (`NP00O00`), and Others (`NP00V00`).

The models can be retrained using an annotated corpus and the scripts in `src/utilities/nerc`.

See file `src/utilities/train-nerc/README` and the comments inside the script for details on how to retrain the models or about the meaning of each file.

The most important file in the set is the `.rgf` file, which contains a definition of the context features that must be extracted for each named entity. The feature rule language is described in section Feature Extractor.

The API of the class is the following:

```
class nec {
  public:
    /// Constructor
    nec(const std::string &cfgfile);

    /// analyze given sentence.
    void analyze(sentence &s) const;

    /// analyze given sentences.
    void analyze(std::list<sentence> &ls) const;

    /// return analyzed copy of given sentence
    sentence analyze(const sentence &s) const;

    /// return analyzed copy of given sentences
    std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor receives one parameter with the name of the configuration file for the module. Its content is described below.

NEC Data File

The machine-learning based Named Entity Classification module reads a configuration file with the following sections

- Section `<RGF>` contains one line with the path to the RGF file of the model. This file is the definition of the features that will be taken into account for NEC.

  ```
  <RGF>
  ner.rgf
  </RGF>
  ```

- Section `<Classifier>` contains one line with the kind of classifier to use. Valid values are `AdaBoost` and `SVM`.

```
<Classifier>
Adaboost
</Classifier>
```

- Section `<ModelFile>` contains one line with the path to the model file to be used. The model file must match the classifier type given in section `<Classifier>`.

```
<ModelFile>
ner.abm
</ModelFile>
```

The `.abm<` files contain AdaBoost models based on shallow Decision Trees (see [CMP03] for details). You don't need to understand this, unless you want to enter into the code of the AdaBoost classifier.

The `.svm` files contain Support Vector Machine models generated by `libsvm` [CL11]. You don't need to understand this, unless you want to enter into the code of `libsvm`.

- Section `<Lexicon>` contains one line with the path to the lexicon file of the learnt model. The lexicon is used to translate string-encoded features generated by the feature extractor to integer-encoded features needed by the classifier. The lexicon file is generated at training time.

```
<Lexicon>
ner.lex
</Lexicon>
```

The `.lex` file is a dictionary that assigns a number to each symbolic feature used in the AdaBoost or SVM model. You don't need to understand this either unless you are a Machine Learning student or the like.

- Section `<Classes>` contains only one line with the classes of the model and its translation to `B`, `I`, `O` tag.

```
<Classes>
0 NP00SP0 1 NP00G00 2 NP00O00 3 NP00V00
</Classes>
```

- Section `<NE_Tag>` contains only one line with the PoS tag assigned by the NER module, which will be used to select named entities to be classified.

```
<NE_Tag>
NP00000
</NE_Tag>
```

Chart Parser Module

The chart parser enriches each `sentence` object with a `parse_tree` object, whose leaves have a link to the sentence words.

The API of the parser is:

```
class chart_parser {
 public:
   /// Constructor
   chart_parser(const std::string &cfgfile);

   /// Get the start symbol of the grammar
   std::string get_start_symbol() const;

   /// analyze given sentence.
   void analyze(sentence &s) const;

   /// analyze given sentences.
   void analyze(std::list<sentence> &ls) const;

   /// return analyzed copy of given sentence
   sentence analyze(const sentence &s) const;

   /// return analyzed copy of given sentences
   std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor receives a file with the CFG grammar to be used by the parser. See below for details.

The method `get_start_symbol` returns the initial symbol of the grammar, and is needed by the dependency parser.

Shallow Parser CFG file

This file contains a CFG grammar for the chart parser, and some directives to control which chart edges are selected to build the final tree. Comments may be introduced in the file, starting with ``%", the comment will finish at the end of the line.

Grammar rules have the form:

```
x ==> y, A, B.
```

That is, the head of the rule is a non-terminal specified at the left hand side of the arrow symbol. The body of the rule is a sequence of terminals and nonterminals separated with commas and ended with a dot.

Empty rules are not allowed, since they dramatically slow chart parsers. Nevertheless, any grammar may be written without empty rules (assuming you are not going to accept empty sentences).

Rules with the same head may be or-ed using the bar symbol, as in:

```
x ==> A, y | B, C.
```

The head component for the rule maybe specified prefixing it with a plus (+) sign, e.g.:

```
nounphrase ==> DT, ADJ, +N, prepphrase.
```

If the head is not specified, the first symbol on the right hand side is assumed to be the head. The head marks are not used in the chart parsing module, but are necessary for later dependency tree building.

The grammar is case-sensitive, so make sure to write your terminals (PoS tags) exactly as they are output by the tagger. Also, make sure that you capitalize your non-terminals in the same way everywhere they appear.

Terminals are PoS tags, but some variations are allowed for flexibility:

- Plain tag: A terminal may be a plain complete PoS tag, e.g. `VMIP3S0`
- Wildcarding: A terminal may be a PoS tag prefix, right-wilcarded, e.g. `VMI*`, `VMIP*`.
- Specifying lemma: A terminal may be a PoS tag (or a wilcarded prefix) with a lemma enclosed in angle brackets, e.g `VMIP3S0<comer>`, `VMI*<comer>` will match only words with those tag/prefix and lemma.
- Specifying form: A terminal may be a PoS tag (or a wilcarded prefix) with a form enclosed in parenthesis, e.g `VMIP3S0(comió)`, `VMI*(comió)` will match only words with those tag/prefix and form.

- If a double-quoted string is given inside the angle brackets or parenthesis (e.g. `VMI*("myforms.dat")` or `VMIP3S0<"mylemmas.dat">`) it is interpreted as a file name, and the terminal will match any word form (or lemma) found in that file. If the file name is not an absolute path, it is interpreted as a relative path based at the location of the grammar file.

The grammar file may contain also some directives to help the parser decide which chart edges must be selected to build the tree. Directive commands start with the directive name (always prefixed with `@`), followed by one or more non-terminal symbols, separated with spaces. The list must end with a dot. Available directives are:

- `@NOTOP`: Non-terminal symbols listed under this directive will not be considered as valid tree roots, even if they cover the complete sentence.
- `@START`: Specify which is the start symbol of the grammar. Exactly one non-terminal must be specified under this directive. The parser will attempt to build a tree with this symbol as a root. If the result of the parsing is not a complete tree, or no valid root nodes are found, a fictitious root node is created with this label, and all created trees are attached to it.
- `@FLAT`: Subtrees for non-terminal symbols specified in this list are flattened when the symbol is recursive. Only the highest occurrence appears in the final parse tree.
- @HIDDEN: Non-teminal symbols specified under this directive will not appear in the final parse tree (their descendant nodes will be attached to their parent).
- @PRIOR: lists of non-terminal symbols in decreasing priority order (the later in the list, the lower priority). When a chart top cell can be covered with two different non-terminals, the one with highest priority is chosen. This has no effect on non-top cells (in fact, if you want that, your grammar is probably ambiguous and you should rethink it...)

Rule-based Dependency Parser Module

The Txala dependency parser [ACM05] gets constituency parsed sentences -that is, `sentence` objects which have been enriched with a `parse_tree` by the `chart_parser` (or by any other means). The input parsing may be shallow. The dependency parser will complete the parse tree if needed, convert it to a dependency tree, and assign a syntactic function to each edge in the dependency tree.

```
class dep_txala : public dependency_parser {
 public:
   /// constructor
   dep_txala(const std::string &cfgfile, const std::string
&start);

   /// analyze given sentence.
   void analyze(sentence &s) const;

   /// analyze given sentences.
   void analyze(std::list<sentence> &ls) const;

   /// return analyzed copy of given sentence
   sentence analyze(const sentence &s) const;

   /// return analyzed copy of given sentences
   std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor receives two strings: the name of the file containging the dependency rules to be used, and the start symbol of the grammar used by the `chart_parser` to parse the sentence.

The dependency parser works in three stages:

- At the first stage, the `<GRPAR>` rules are used to complete the shallow parsing produced by the chart into a complete parsing tree. The rules are applied to a pair of adjacent chunks. At each step, the selected pair is fused in a single chunk. The process stops when only one chunk remains.
- The next step is an automatic conversion of the complete parse tree to a dependency

tree. Since the parsing grammar encodes information about the head of each rule, the conversion is straighforward.
- The last step is the labeling. Each edge in the dependeny tree is labeled with a syntactic function, using the `<GRLAB>` rules.

The syntax and semantics of `<GRPAR>` and `<GRLAB>` rules are described below.

Dependency Parsing Rule File

The dependency rules file contains a set of rules to perform dependency parsing.

The file consists of five sections: sections: `<GRPAR>`, `<GRLAB>`, `<SEMDB>`, `<CLASS>`, and `<PAIRS>`.

Parse-tree completion rules

Section `<GRPAR>` contains rules to complete the partial parsing provided by the chart parser. The tree is completed by combining chunk pairs as stated by the rules. Rules are applied from highest priority (lower values) to lowest priority (higher values), and left-to right. That is, the pair of adjacent chunks matching the most prioritary rule is found, and the rule is applied, joining both chunks in one. The process is repeated until only one chunk is left.

The rules can be enabled/disabled via the activation of global flags. Each rule may be stated to be enabled only if certain flags are on. If none of its enabling flags are on, the rule is not applied. Each rule may also state which flags have to be toggled on/off after its application, thus enabling/disabling other rule subsets.

Each line in section `<GRPAR>` contains a rule, with the format:
`priority flags context (lchunk,rchunk) pair-constraints operation op-params flag-ops`

where:

- `priority` is a number stating the priority of a rule (the lower the number, the higher the priority).
- `flags` is a list of strings separated by vertical bars (`|`). Each string is the name of a flag that will cause the rule to be enabled. If this field is `-`, the rule will be always enabled.
- `context` is a context limiting the application of the rule only to chunk pairs that are surrounded by the appropriate context. A dash (`-`) means no restrictions, and the rule is applied to any matching chunk pair.
- `(lchunk,rchunk)` are the labels of the adjacent pair of chunks the rule may be applied to. The labels are either assigned by the chunk parser, or by a `RELABEL` operation on some other completion rule. The pair must be enclosed in parenthesis, separated by a

comma, and contain NO whitespaces.

The chunk labels may be suffixed with one extra condition of the form: `(form)`, `<lemma>`, `[class]`, or `{PoS_regex}`.

For instance,

The label:	Would match:
`np`	any chunk labeled `np` by the chunker
`np(cats)`	any chunk labeled `np` by the chunker with a head word with form `cats`
`np<cat>`	any chunk labeled `np` by the chunker with a head word with lemma `cat`
`np[animal]`	any chunk labeled `np` by the chunker with a head word with a lemma in `animal` category (see `CLASS` section below)
`np{^N.M[PS]}`	any chunk labeled `np` by the chunker with a head word with a PoS tag matching the regular expression `^N.M[PS]`

- `pair-constraits` expresses a constraint that must be satisfied by the pair of target chunks `(lchunk,rchunk)`. If no constraints are required, this field must be a dash (`-`). The format of the constraint is `pairclass::(value1,value2)`, where:
  - `pairclass` is the name of a pair class defined in the `<PAIRS>` section (see below).
  - `value1` and `value2` are the two values that must belong to the pair class.

    Each `value` specifies whether the value is to be extracted from `lchunk` (`L`) or `rchunk` (`R`), the path to or a node below them (if target is not the root), and the attribute to extract.

    For instance, `L.lemma` specifies the lemma of the head word of `lchunk`. `R:sn.pos` specifies the PoS tag of the head word of a node with label `sn` located under `rchunk`. `R:sp:sn.semfile` specifies the semantic file of the head word of a node with label `sn` located under a node with label `sp` located under `rchunk`.

    Valid attributes are: `lemma`, `pos`, `semfile`, `tonto`, `synon`, `asynon`. Their meaning is the same than for dependency labeling rules, and is described below.
- `operation` is the way in which `lchunk` and `rchunk` nodes are to be combined (see below).
- `op-params` has two possible meanings, depending on the `operation` field: `top_left` and `top_right` operations must be followed by the literal `RELABEL` plus the new label(s) to assign to the chunks. Other operations must be followed by the literal `MATCHING` plus the label to be matched.

For `top_left` and `top_right` operations the labels following the keyword `RELABEL` state the labels with which each chunk in the pair will be relabelled, in the format `label1:label2`. If specified, `label1` will be the new label for the left chunk, and `label2` the one for the right chunk. A dash (`-`) means no relabelling. In none of both chunks is to be relabelled, `-` may be used instead of `-:-`. For example, the rule: `20 - - (np,pp<of>) top_left RELABEL np-of:- -`
will hang the `pp` chunk as a daughter of the left chunk in the pair (i.e. `np`), then relabel the `np` to `np-of`, and leave the label for the `pp` unchanged.

For `last_left`, `last_right` and `cover_last_left` operations, the label following the keyword `MATCHING` states the label that a node must have in order to be considered a valid `last` and get the subtree as a new child. This label may carry the same modifying suffixes than the chunk labels. If no node with this label is found in the tree, the rule is not applied. For example, the rule:
`20 - - (vp,pp<of>) last_left MATCHING np -`
will hang the `pp` chunk as a daughter of the last subtree labeled `np` found inside the `vp` chunk.

- The last field `flag-ops` is a space-separated list of flags to be toggled on/off. The list may be empty (meaning that the rule doesn't change the status of any flag). If a flag name is preceded by a `+`, it will be toggled on. If the leading symbol is a `-`, it will be toggled off.

For instance, the rule:

```
20 - - (np,pp<of>) top_left RELABEL - -
```

states that if two subtrees labelled `np` and `pp` are found contiguous in the partial tree, and the second head word has lemma `of`, then the later (rightmost) is added as a new child of the former (leftmost), whatever the context is, without need of any special flag active, performing no relabelling of the new tree root, and without activating or deactivating any flags.

The supported tree-building operations are the following:

- `top_left`: The right subtree is added as a daughter of the left subtree. The root of the new tree is the root of the left subtree. If a `label` value other than `-` is specified, the root is relabelled with that string.
- `last_left`: The right subtree is added as a daughter of the last node inside the left subtree matching `label` value (or to the root if none is found). The root of the new tree is the root of the left subtree.
- `top_right`: The left subtree is added as a new daughter of the right subtree. The root of the new tree is the root of the right subtree. If a `label` value other than `-` is

specified, the root is relabelled with that string.

- `last_right` : The left subtree is added as a daughter of the last node inside the right subtree matching `label` value (or to the root if none is found). The root of the new tree is the root of the right subtree.
- `cover_last_left` : The left subtree (*s*) takes the position of the last node (*x*) inside the right subtree matching `label` value. The node *x* is hanged as new child of *s*. The root of the new tree is the root of the right subtree.

The context may be specified as a sequence of chunk labels, separated by underscores `_` . One of the chunk labels must be `$$` , and refers to the pair of chunks which the rule is being applied to.

For instance, the rule:

```
20 - $$_vp (np,pp<of>) top_left RELABEL -
```

would add the right chunk in the pair (`pp<of>`) under the left one (`np`) only if the chunk immediate to the right of the pair is labeled `vp` .

Other admitted labels in the context are: `?` (matching exactly one chunk, with any label), `*` (matching zero or more chunks with any label), and `OUT` (matching a sentence boundary).

For instance the context `np_$$_*_vp_?_OUT` would match a sentence in which the focus pair of chunks is immediately after an `np` , and the second-to-last chunk in the sentence is labeled `vp` .

Context conditions can be globally negated preceding them with an exclamation mark (`!`). E.g. `!np_$$_*_vp` would cause the rule to be applied only if that particular context is not satisfied.

Context condition components may also be individually negated preceding them with the symbol `~` . E.g. the rule `np_$$_~vp` would be satisfied if the preceding chunk is labeled `np` and the following chunk has any label but `vp` .

Enabling flags may be defined and used at the grammarian's will. For instance, the rule:

```
20 INIT|PH1 $_vp (np,pp<of>) last_left MATCHING nphrase[animal]
+PH2 -INIT -PH1
```

Will be applied if either `INIT` or `PH1` flags are on, the chunk pair is a `np` followed by a `pp` with head lemma `of` , and the context (one `vp` chunk following the pair) is satisfied. Then, the deepest rightmost node matching the label `nphrase[animal]` will be sought in the

left chunk, and the right chunk will be linked as one of its children. If no such node is found, the rule will not be applied.

After applying the rule, the flag `PH2` will be toggled on, and the flags `INIT` and `PH1` will be toggled off.

The only predefined flag is `INIT`, which is toggled on when the parsing starts. The grammarian can define any alphanumerical string as a flag, simply toggling it on in some rule.

Dependency function labeling rules

Labelling rules defined in section `<GRLAB>` are applied once the tree has been completed and converted to a dependency tree. For each edge in the tree, the first matching rule is located and applied.

Section `<GRLAB>` contains two kind of lines.

The first kind are the lines defining `UNIQUE` labels, which have the format:

```
UNIQUE label1 label2 label3 ...
```

You can specify many `UNIQUE` lines, each with one or more labels. The effect is the same than having all of them in a single line, and the order is not relevant.

Labels in `UNIQUE` lists will be assigned only once per head. That is, if a head has a daugther with a dependency already labeled as `label1`, rules assigning this label will be ignored for all other daugthers of the same head. (e.g. if a verb has got a `subject` label for one of its dependencies, no other dependency will get that label, even if it meets the conditions to do so).

The second kind of lines state the rules to label the dependences extracted from the full parse tree build with the rules in previous section:

Each line contains a rule, with the format:

```
ancestor-label dependence-label condition1 condition2 ...
```

where:

- `ancestor-label` is the label of the node which is head of the dependence.
- `dependence-label` is the label to be assigned to the dependence
- `condition` is a list of conditions that the dependence has to match to satisfy the rule.

Each `condition` has one of the forms:

```
node.attribute = value
node.attribute != value
```

Where `node` is a string describing a node on which the `attribute` has to be checked. The `value` is a string to be matched, or a set of strings (separated by `|`). The strings can be right-wildcarded (e.g. `np*` is allowed, but not `n*p`). For the `pos` attribute, `value` can be any valid regular expression.

The `node` expresses a path to locate the node to be checked. The path must start with `p` (parent node) or `d` (descendant node), and may be followed by a colon-separated list of labels. For instance `p:sn:n` refers to the first node labeled `n` found under a node labeled `sn` which is under the dependency parent `p`.

The `node` may be also `As` (*All siblings*) or `Es` (*Exists sibling*) which will check the list of all children of the ancestor (`p`), excluding the focus daughter (`d`). `As` and `Es` may be followed by a path, just like `p` and `d`. For instance, `Es:sn:n` will check for a sibling with that path, and `As:sn:n` will check that all siblings have that path.

The `node` may be also a pair of the form `[node1.attribute,node2.attribute]` where `node1` and `node2` can be a node path (e.g. `p:sn:n`, `p`, etc) but not `As`, `Es`, or another pair. In this case, the attributes for `node1` and `node2` can only be one of `label`, `lemma`, or `pos`. Finally, the attribute for such a pair node can be only `pairclass`.

Possible `attribute` to be used:

- `label`: chunk label (or PoS tag) of the node.
- `side`: (left or right) position of the specified node with respect to the other. Only valid for `p` and `d`.
- `lemma`: lemma of the node head word.
- `pos`: PoS tag of the node head word
- `class`: word class (see below) of lemma of the node head word.
- `tonto`: EWN Top Ontology properties of the node head word.
- `semfile`: WN semantic file of the node head word.
- `synon`: Synonym lemmas of the node head word (according to WN).
- `asynon`: Synonym lemmas of the node head word ancestors (according to WN).
- `pairclass`: Only applicable to pair nodes. Check whether the pair is in any of specified classes.

Note that since no disambiguation is required, the attributes dealing with semantic properties will be satisfied if any of the word senses matches the condition.

For instance, the rule:

```
verb-phr    subj    d.label=np*      d.side=left
```

states that if a `verb-phr` node has a daughter to its left, with a label starting by `np`, this dependence is to be labeled as `subj`.

Similarly, the rule:

```
verb-phr    obj    d.label=np*  d:sn.tonto=Edible
p.lemma=eat|gulp
```

states that if a `verb-phr` node has `eat` or `gulp` as lemma, and a descendant with a label starting by `np` and containing a daughter labeled `sn` that has `Edible` property in EWN Top ontology, this dependence is to be labeled as `obj`.

Another example:

```
verb-phr    iobj   d.label=pp* d.lemma=to|for  Es.label=np*
```

states that if a `verb-phr` has a descendant with a label starting by `pp` (prepositional phrase) and lemma `to` or `for`, and there is another child of the same parent which is a noun phrase (`np*`), this dependence is to be labeled as `iobj`.

Yet another:

```
verb-phr    dobj   d.label=pp* d.lemma=to|for  As.label!=np*
```

states that if a `verb-phr` has a descendant with a label starting by `pp` (prepositional phrase) and lemma `to` or `for`, and all the other children of the same parent are not noun phrases (`np*`), this dependence is to be labeled as `dobj`.

And some examples for `pairclass`:

```
verb-phr dobj d.label=noun-phr* d.side=right
[p.lemma,d.lemma].pairclass=direct
verb-phr loc d.label=pp* d.lemma=in|at d.side=right
[p.lemma,d:noun-phr.lemma].pairclass=location
```

First rule above states that a `noun-phr` daughter under a `verb-phr` parent will be labeled as `dobj` if it is to the right of its parent, and the pair formed by their lemmas is found in the `direct` pair class (which should be defined in section `<PAIRS>` as described below).

The second rule states that a `pp` daughter under a `verb-phr` parent will be labeled as `loc` if it is to the right of its parent, the preposition heading the `pp` is `in` or `at`, and the pair formed by the verb lemma and the noun heading the noun phrase inside the `pp` is found in the `location` pair class (which should be defined in section `<PAIRS>` as described below).

Semantic database location

Section `<SEMDB>` is only necessary if the dependency labeling rules in section `<GRLAB>` use conditions on semantic values (that is, any of `tonto`, `semfile`, `synon`, or `asynon`). Since it is needed by `<GRLAB>` rules, section `<SEMDB>` must be defined before section `<GRLAB>`. The section must contain a single line specifying a configuration file for a semanticDB object. The filename may be absolute or relative to the location of the dependency rules file.

```
<SEMDB>
../semdb.dat
</SEMDB>
```

The configuration file must follow the format described in section Semantic Database.

Class definitions

Section `<CLASS>` contains class definitions which may be used as attributes in the dependency labelling rules.

Each line contains a class assignation for a lemma, with two possible formats:

```
class-name  lemma      comments
class-name  "filename"   comments
```

For instance, the following lines assign to the class `mov` the four listed verbs, and to the class `animal` all lemmas found in `animals.dat` file. In the later case, if the file name is not an absolute path, it is interpreted as a relative path based at the location of the rule file.

Anything to the right of the second field is considered a comment and ignored.

```
<CLASS>
mov     go      %%prep= to,towards  (note that "%%" is not
required for comments)
mov     come    %%prep= from        (anything after 2nd field is
ignored)
mov     walk    %%prep= through
mov     run     %%prep= to,towards   D.Obj.

animal "animals.dat"
</CLASS>
```

Pair-class definitions

Section `<PAIRS>` contains class definitions of compatible pairs. They can be used as attributes in the tree-completing rules.

Each line contains a class assignation for a pair, with two possible formats:

```
class-name  element1 element2
class-name  "filename"
```

For instance, the following lines assign to the class `material` the two first pairs, to the class `location` the third and fourth pair, and to class `food"` all pairs found in `food.dat` file. In the later case, if the file name is not an absolute path, it is interpreted as a relative path based at the location of the rule file. The contents of the file must be a list of pairs (one per line).

No comments are allowed in the same line than a pair.

```
<PAIRS>
material   table wood
material   knife steel
location   tree forest
location   car road

food    "food.dat"
</PAIRS>
```

These pair classes may be used by tree-completion rules to set constraints to check whether two chunks must be joined or not, and by labeling rules to check whether a head and its dependant have a certain type of relation.

Statistical Dependency Parser and Semantic Role Labelling Module

As an alternative to rule-based Txala dependency parser, a statistical dependency parsing module is also available. It is based on Treeler machine learning library.

The dependency parser is based on the paper [Car07], and the Semantic Role Labelling module follows the proposal by [LCM].

The API of the class is the following:

```
class dep_treeler : public dependency_parser {
 public:
   /// constructor
   dep_treeler(const std::string &cfgfile);
   /// destructor
   ~dep_treeler();

   /// analyze given sentence.
   void analyze(sentence &s) const;

   /// analyze given sentences.
   void analyze(std::list<sentence> &ls) const;

   /// return analyzed copy of given sentence
   sentence analyze(const sentence &s) const;

   /// return analyzed copy of given sentences
   std::list<sentence> analyze(const std::list<sentence> &ls)
const;
};
```

The constructor for class `dep_treeler` expects a configuration file with the contents described below.

The module can perform not only dependency parsing, but also semantic role labelling (SRL) if it has been trained to do so.

The configuration file establishes whether both tasks are performed, or just dependency parsing.

Statistical Parser and SRL Configuration File

The configuration file for the statistical dependency parser and semantic role labelling module has two main sections: `<Dependencies>` and `<SRL>`. Each section establishes the configuration and parameters of the corresponding subtask. Section `<Dependencies>` is required, but section `<SRL>` may be ommitted if no SRL is required.

Section `<Dependencies>` contains two lines, with a keyword and a value each:

- The `DependencyTreeler` keyword should be followed by a path to a Treeler configuration file with the dependency parsing model to use. The path may be either absolute or relative to the Statistical Parser and SRL configuration file.
- The `Tagset` keyword should be followed by a path to a tagset definition file which will be used to convert the input PoS tags to the short versions and MSD features expected by the Treeler model.

An example of the `<Dependencies>` section:

```
<Dependencies>
## treeler config file for dep parser
DependencyTreeler ./dep/config.dat

## Tagset description file
Tagset ./tagset.dat
</Dependencies>
```

Section `<SRL>` contains also keyword-value lines, which may be of 4 different types:

- `Predicates`: Lines stating a PoS tag and a list of files containing lines with the format: `sense predicate argument1 argument2 ... [*]`
  E.g.

  ```
  00028565-v smile.01 A0:Agent A1:Theme A2:Recipient
  00008435-v wink.01|blink.01 A0:Agent A1:Patient A2:Recipient
  A3:Theme
  00014201-v shudder.01|shiver.01 A1:Experiencer A2:Stimulus
  ```

If the word sense is contained in one of the lists, the word is considered as a predicate for SRL, and its sense number and argument pattern are retrieved from the list.

An asterisk may be added to the end of the line indicating that words not found in any list, but with the appropriate PoS, should be considered predicates too.

- `PredicateException` : This lines list specific cases of words that should not be considered predicates, even if they satisfy some of the conditions of a `Predicates` line. An additional PoS may be added, meaning that the exception holds only if the target word has a dependant with that PoS. This is typically used to exclude auxiliary verbs from being considered predicates (e.g. if there is a `Predicates` rule that would consider any verb as a predicate, auxiliary verbs that have another verb as dependant are not to be considered as predicates).
- `DefaultArgs` : List of labels for the argument frame to be used for predicates not found in any list (but considered predicates because some `Predicates` line had an asterisk).
- `SRLTreeler` : This keyword must be followed by a path to a Treeler configuration file with the SRL model to use. The path may be either absolute or relative to the Statistical Parser and SRL configuration file.

An example of the `<SRL>` section:

```
<SRL>
## Files containing conversion synset->predicate->argument list
## Each PoS admits a list of files, checked in cascade.
## Asterisc means any word with that PoS will be a predicate,
## even if it is not in any list. In that case, predicate
number
## will be ``.00'' and arguments will be those in
``DefaultArgs''
Predicates V ../../common/pred-verb.dat *
Predicates N ../../common/pred-nom.dat

## Execptions: words that are NOT predicates, even if they are
## accepted by rules above (e.g. matched an asterisc).
## The execption holds only if the word has a daugher with given
PoS.
PredicateException V:have V
PredicateException V:be V
PredicateException V:do V

## arguments for predicates not found in the list (e.g. accepted
## by an asterisc)
DefaultArgs A0 A1 A2 A3

## If you do not need SRL, comment out SRLTreeler line or remove
section <SRL>
## treeler config file for SRL
SRLTreeler ./srl/config.dat
</SRL>
```

RelaxCor Coreference Resolution Module

This module offers coreference resolution capabilities.

It is based on the research described in [Sap13], the second-ranked system at CoNLL 2011 shared task.

The API to this module is:

```
class relaxcor {
  public:

    /// Constructor
    relaxcor(const std::wstring &fconfig);
    /// Destructor
    ~relaxcor();

   /// Finds the coreferent mentions in a document
    void analyze(document&) const;
  };
```

The module constructor expects a configuration file that define which resources to load and parameters to use. This file also defines which are the configuration files for several sub-modules (mention detection, feature extraction, etc).

Once created, the module can be given a document (which in general should be fully analyzed with shallow parsing, dependency parsing, NEC, WSD, and SRL) and it will enrich the document metadata with coreference information.

Note that this module can only process whole documents, and not single sentences or paragraphs as the other modules, since coreference is a document-level phenomenon.

Semantic Graph Extractor Module

This module will combine the information provided by the parser, the SRL, and the coreference resolution modules, and build a semantic graph encoding which events are described in the text, which are the relations between them, and which are the actors participating in those events.

```
class semgraph_extract {

  public:
    /// Constructor
    semgraph_extract(const std::wstring &CfgFile);
    /// Destructor
    ~semgraph_extract();

    /// extract graph from given document
    void extract(freeling::document &doc) const;

  };
```

This module will enrich the document with an object `semantic_graph` containing a pseudo entity-relationship model of the events described in the text.

For instance, for the text: *Mary says that Peter Smith bought a car that runs fast. Peter loves Mary.* FreeLing would produce the following graph:

```
<semantic_graph>
   <entity id="E1" lemma="mary" class="person">
      <mention id="t1.1" words="Mary" />
      <mention id="t2.3" words="Mary" />
   </entity>
   <entity id="E2" lemma="peter_smith" class="person" >
      <mention id="t1.4" words="Peter_Smith" />
      <mention id="t2.1" words="Peter" />
   </entity>
   <entity id="W6" lemma="car" sense="02958343-n" >
      <mention id="t1.7" words="a car that runs fast" />
      <synonym lemma="auto"/>
```

```
      <synonym lemma="automobile"/>
      <synonym lemma="car"/>
      <synonym lemma="machine"/>
      <synonym lemma="motorcar"/>
      <URI knowledgeBase="WordNet"
           URI="http://wordnet-rdf.princeton.edu/wn30/02958343-
n"/>
      <URI knowledgeBase="OpenCYC"

URI="http://sw.opencyc.org/concept/Mx4rvViVwZwpEbGdrcN5Y29ycA"/>
      <URI knowledgeBase="SUMO"
           URI="http://ontologyportal.org/SUMO.owl#Automobile"/>
   </entity>
   <entity id="W8" lemma="fast" sense="00086000-r" >
      <mention id="t1.10" words="fast" />
      <synonym lemma="fast"/>
      <URI knowledgeBase="WordNet"
           URI="http://wordnet-rdf.princeton.edu/wn30/00086000-
r"/>
      <URI knowledgeBase="SUMO"

URI="http://ontologyportal.org/SUMO.owl#SubjectiveAssessmentAttr
ibute"/>
   </entity>
   <frame id="F3" token="t1.2" lemma="say.00" sense="00928959-v"
>
      <argument role="A0" entity="E1" />
      <argument role="A1" entity="F4" />
      <synonym lemma="say"/>
      <URI knowledgeBase="WordNet"
           URI="http://wordnet-rdf.princeton.edu/wn30/00928959-
v"/>
      <URI knowledgeBase="SUMO"
           URI="http://ontologyportal.org/SUMO.owl#Process"/>
   </frame>
   <frame id="F4" token="t1.5" lemma="purchase.01|buy.01"
sense="02207206-v" >
      <argument role="A0:Agent" entity="E2" />
      <argument role="A1:Theme" entity="W6" />
      <synonym lemma="buy"/>
```

```
      <synonym lemma="purchase"/>
      <URI knowledgeBase="WordNet"
           URI="http://wordnet-rdf.princeton.edu/wn30/02207206-
v"/>
      <URI knowledgeBase="OpenCYC"

URI="http://sw.opencyc.org/concept/Mx4rvVjL2pwpEbGdrcN5Y29ycA"/>
      <URI knowledgeBase="SUMO"
           URI="http://ontologyportal.org/SUMO.owl#Buying"/>
   </frame>
   <frame id="F5" token="t1.9" lemma="run.00" sense="01926311-v"
>
      <argument role="A0" entity="W6" />
      <argument role="AM-MNR" entity="W8" />
      <synonym lemma="run"/>
      <URI knowledgeBase="WordNet"
           URI="http://wordnet-rdf.princeton.edu/wn30/01926311-
v"/>
      <URI knowledgeBase="OpenCYC"

URI="http://sw.opencyc.org/concept/Mx4rvVjkGpwpEbGdrcN5Y29ycA"/>
      <URI knowledgeBase="SUMO"
           URI="http://ontologyportal.org/SUMO.owl#Running"/>
   </frame>
   <frame id="F9" token="t2.2" lemma="love.01" sense="01775164-
v" >
      <argument role="A0:Experiencer" entity="E2" />
      <argument role="A1:Stimulus" entity="E1" />
      <synonym lemma="love"/>
      <URI knowledgeBase="WordNet"
           URI="http://wordnet-rdf.princeton.edu/wn30/01775164-
v"/>
      <URI knowledgeBase="SUMO"
           URI="http://ontologyportal.org/SUMO.owl#wants"/>
   </frame>
</semantic_graph>
```

The graph encodes that there are three entities in the text (*Peter*, *Mary*, and *a car*). Some of them are mentioned more than once (and so, they participate in different events described in the text). Some of them have a class (e.g. `person`) or a link to an external ontology (such as

WN, SUMO, or OpenCYC).

Then, there are some events (or `frames`) described in the text. Those frames have arguments, which are the actors involved in each event. External URIs for the semantic of those actions are also provided.

So the main relations depicted in this example graph are:

- Frame `F9` describes a `love.01` event experienced by entity `E2` (*Mary*) due to the stimulus of entity `E1` (*Peter*).
- Frame `F5` describes a `run.00` event performed by entity `W6` (*a car*). This event has a manner argument that is `W8` (*fast*).
- Frame `F4` describes a `purchase.01|buy.01` event performed by `E2` (*Peter*) on `W6` (*a car*).
- Frame `F3` describes a `say.00` event where `E1` (*Mary*) states that event `F4` (Peter buying a car) happened.

Analyzer Metamodule

Although FreeLing is a toolbox with a variety of modules to pick and choose from for specific uses, most applications are likely to need a text analisys pipeline that goes from raw text to certain level of annotation.

To ease the construction of applications that call FreeLing, the `analyzer` metamodule has been included. This metamodule implements a pipeline calling a standard sequence of FreeLing modules: Tokenizer, splitter, morphological analyzer, PoS tagger, sense annotation, WSD, NEC, parsing, SRL, coreferences.

A set of customizable options in this module allows the calling application to control the start and ending levels of the pipeline (e.g. from text to shallow parsing, or from tagged input to coreferences...), as well as which modules are turned on/off and with which configuration files are they loaded.

Thus, different instances of this class can be created using different option sets to get analyzers for different languages or different tasks.

The API of the class is the following:

```
class analyzer {
 public:
   typedef analyzer_config_options config_options;
   typedef analyzer_invoke_options invoke_options;

   /// constructor, given a set of creation options
   analyzer(const config_options &cfg);

   /// Destructor
   ~analyzer();

   /// get current execution options
   const invoke_options& get_current_invoke_options() const;

   /// change execution options for next call
   void set_current_invoke_options(const invoke_options &opt,
bool check=true);

   /// analyze text as a whole document.
```

```
   /// 'parag' indicates whether a blank line is to be
considered a paragraph
   /// separator.
   void analyze(const wstring &text, document &doc, bool
parag=false) const;

   /// Analyze text as a partial document. Retain incomplete
sentences in buffer
   /// in case next call completes them (except if flush==true)
   void analyze(const wstring &text, std::list<sentence> &ls,
bool flush=false);

   /// analyze further levels on a partially analyzed document
   void analyze(document &doc) const;

   /// analyze further levels on partially analyzed sentences
   void analyze(std::list<sentence> &ls) const;

   /// flush splitter buffer and analyze any pending text.
   void flush_buffer(std::list<sentence> &ls);

   /// Reset tokenizer byte offset counter to 0.
   void reset_offset();
};
```

The constructor expects a set of configuration options (see `class analyzer::config_options` below) which specifiy creation-time options for all modules that need to be loaded. These options are basically configuration and data files to load.

The `analyzer` meta-module will create all modules for which a configuration file is specified. If a module does not need to be created, the corresponding option in `analyzer::config_options` should be empty.

Once the `analyzer` instance is created, a set of invocation options must be specified (see description of class `analyzer::invoke_options` below) . Invocation options are run-time options and can be altered for each analysis if necessary (e.g. if one needs to apply different processes to different kinds of input texts). They include activating/deactivating modules or changing the initial/final points in the pipeline.

When invoke options are set, the `analyzer` meta-module can be called to process a plain text, or to enrich a partially analyzed document.

Analyzer configuration options

Class `analyzer::config_options` contains the configuration options that define which modules are active and which configuration files are loaded for each of them at construction time. Options in this set can not be altered once the analyzer is created. If an option has an empty value, the corresponding module will not be created (and thus it will not be possible to call it just altering `invoke_options` later)

```
class analyzer::config_options {
  public:
     /// Language of text to process
     std::wstring Lang;

     /// Tokenizer configuration file
     std::wstring TOK_TokenizerFile;

     /// Splitter configuration file
     std::wstring SPLIT_SplitterFile;

     /// Morphological analyzer options
     std::wstring MACO_Decimal, MACO_Thousand;
     std::wstring MACO_UserMapFile, MACO_LocutionsFile,
MACO_QuantitiesFile,
                  MACO_AffixFile,   MACO_ProbabilityFile,
MACO_DictionaryFile,
                  MACO_NPDataFile,  MACO_PunctuationFile,
MACO_CompoundFile;
     double MACO_ProbabilityThreshold;

     /// Phonetics config file
     std::wstring PHON_PhoneticsFile;

     /// NEC config file
     std::wstring NEC_NECFile;

     /// Sense annotator and WSD config files
     std::wstring SENSE_ConfigFile;
     std::wstring UKB_ConfigFile;

     /// Tagger options
```

```
        std::wstring TAGGER_HMMFile;
        std::wstring TAGGER_RelaxFile;
        int TAGGER_RelaxMaxIter;
        double TAGGER_RelaxScaleFactor;
        double TAGGER_RelaxEpsilon;
        bool TAGGER_Retokenize;
        ForceSelectStrategy TAGGER_ForceSelect;

        /// Chart parser config file
        std::wstring PARSER_GrammarFile;

        /// Dependency parsers config files
        std::wstring DEP_TxalaFile;
        std::wstring DEP_TreelerFile;

        /// Coreference resolution config file
        std::wstring COREF_CorefFile;

        /// semantic graph extractor config file
        std::wstring SEMGRAPH_SemGraphFile;
};
```

Analyzer invocation options

Class `analyzer::invoke_options` contains the options that define the behaviour of each module in the analyze on the all subsequent analysis (until invoke options are changed again) Options in this set can be altered after construction (e.g. to activate/deactivate certain modules), as many times as needed.

Values for this options need to be consistent with configuration options used to create the analyzer (e.g. it is not possible to activate a module that has not been loaded at creation time)

```
class analyzer_invoke_options {
   public:
     /// Level of analysis in input and output
     AnalysisLevel InputLevel, OutputLevel;

     /// activate/deactivate morphological analyzer modules
     bool MACO_UserMap, MACO_AffixAnalysis,
MACO_MultiwordsDetection,
          MACO_NumbersDetection, MACO_PunctuationDetection,
          MACO_DatesDetection, MACO_QuantitiesDetection,
          MACO_DictionarySearch, MACO_ProbabilityAssignment,
MACO_CompoundAnalysis,
          MACO_NERecognition, MACO_RetokContractions;

     /// activate/deactivate phonetics
     bool PHON_Phonetics;

     /// activate/deactivate NEC
     bool NEC_NEClassification;

     /// Select which WSD to use (NO_WSD,ALL,MFS,UKB)
     WSDAlgorithm SENSE_WSD_which;

     /// Select which tagger to use (NO_TAGGER,HMM,RELAX)
     TaggerAlgorithm TAGGER_which;

     /// Select which dependency parser to use
(NO_DEP,TXALA,TREELER)
     DependencyParser DEP_which;
};
```

Tag Set Managing Module

This module is able to store information about a tagset, and offers some useful functions on PoS tags and morphological features.

This module is internally used by some analyzers (e.g. probabilities module, HMM tagger, feature extraction, ...) but can be instantiated and called by any user application that requires it.

The API of the module is:

```
class tagset {

  public:
     /// constructor: load a tag set description file
     tagset(const std::wstring &f);
     /// destructor
     ~tagset();

     /// get short version of given tag
     std::wstring get_short_tag(const std::wstring &tag) const;
     /// get map of <feature,value> pairs with morphologica
information for given tag
     std::map<std::wstring,std::wstring>
                 get_msd_features_map(const std::wstring &tag)
const;
     /// get list of <feature,value> pairs with morphologica
information for given tag
     std::list<std::pair<std::wstring,std::wstring> >
           get_msd_features(const std::wstring &tag) const;
     /// get a string with <feature,value> pairs with
morphologica information for given tag
     std::wstring get_msd_string(const std::wstring &tag) const;

     /// convert list of <feature,value> pairs to a PoS tag.
     /// if the list does not contain a value for feature 'pos',
the category must be
     /// scpecified in the 'cat' parameter. Valid categories are
those defined in the
```

```
     /// tagset description file loaded by the constructor.
     std::wstring
        msd_to_tag(const std::wstring &cat,
                   const
std::list<std::pair<std::wstring,std::wstring> > &msd) const;

     /// convert string of <feature,value> pairs to a PoS tag.
     /// if the list does not contain a value for feature 'pos',
the category must be
     /// scpecified in the 'cat' parameter. Valid categories are
those defined in the
     /// tagset description file loaded by the constructor.
     std::wstring msd_to_tag(const std::wstring &cat,const
std::wstring &msd) const;
};
```

The class constructor receives a file name with a tagset description. Format of the file is described below. The class offers three kinds of services:

1. Get the short version of a tag. This is useful for EAGLES tagsets, and required by some modules (e.g. PoS tagger). The length of a short tag is defined in the tagset description file, and depends on the language and part-of-speech. The criteria to select it is usually to have a tag informative enough (capturing relevant features such as category, subcategory, case, etc) but also general enough so that significative statistics for PoS tagging can be acquired from reasonably-sized corpora. For instance, in latin languages the PoS tag for nouns includes gender and number information (e.g. `NCMS000`), but using the whole tag results in statistical dispersion when estimating tagger or parser probabilities. So, the short version `NC` is used. Tagset description file defines which digits should be extracted from the full tag to build the short version.
2. Decompose a tag into a list of pairs feature-value (e.g. `gender=masc`, `num=plural`, `case=dative`, etc). This can be retrieved as a map, as a list of string pairs, or as a formatted string.
3. Given a list of pairs feature-value for morphological attributes, return a PoS tag encoding those properties.

Tagset Description File

Tagset description file has two sections: `<DecompositionRules>` and `<DirectTranslations>`, which describe how tags are converted to their short version and decomposed into morphological feature-value pairs

- Section `<DirectTranslations>` describes a direct mapping from a tag to its short version and to its feature-value pair list. Each line in the section corresponds to a tag, and has the format:
  `tag short-tag feature-value-pairs`

  For instance the line:
  `NCMS000 NC postype=common|gender=masc|number=sing` states that the tag `NCMS000` is shortened as `NC` and that its list of feature-value pairs is the one specified.

  This section has precedence over section `<DecompositionRules>`, and can be used as an exception list. If a tag is found in section `<DirectTranslations>`, the rule is applied and any rule in section `<DecompositionRules>` for this tag is ignored.

- Section `<DecompositionRules>` encodes rules to compute the morphological features from an EAGLES label. The rules describe the possible values and meaning of each position in the label. The form of each line is:
  `tag short-tag-size category position-description-1 position-description-2 ...`
  where `tag` is the character for the category in the EAGLES PoS tag (i.e. the first character: `N`, `V`, `A`, etc.), and `short-tag-size` is an integer stating the length of the short version of the tag (e.g. if the value is 2, the first two characters of the EAGLES PoS tag will we used as short version). `Category` is the name of the main category (e.g. `noun`, `verb`, etc.).

  Finally, fields `position-description-n` contain information on how to interpret each character in the EAGLES PoS tag.
  There should be as many `position-description` fields as characters there are in the PoS tag for that category. Each `position-description` field has the format:
  `feature/char:value;char:value;char:value;...`
  That is: the name of the feature encoded by that character (e.g. `gender`, `number`, etc.) followed by a slash, and then a semicolon-separated list of translation pairs that, for each possible character in that position give the feature value.

  For instance, the rule for Spanish noun PoS tags is (in a single line):
  `N 2 noun type/C:common;P:proper gen/F:fem;M:masc;C:common num/S:sing;P:plur;N:inv neclass/S:person;G:location;O:organization;V:other grade/V:evaluative`
  and states that any tag starting with `N` (unless it is found in section `<DirectTranslations>`) will be shortened using its two first characters (e.g. `NC`, or `NP`). Then, the description of each character in the tag follows, encoding the information:

  1. `type/C:common;P:proper` - second digit is the subcategory (feature `type`) and its

possible values are `C` (translated as `common`) and `P` (translated as `proper`).

2. `gen/F:fem;M:masc;C:common` - third digit is the gender (feature `gen`) and its possible values are `F` (feminine, translated as `fem`), `M` (masculine, translated as `masc`), and `C` (common/invariable, translated as `common`).
3. `num/S:sing;P:plur;N:inv` - fourth digit is the number (feature `num`) and its possible values are `S` (singular, translated as `sing`), `P` (plural, translated as `plur`), and `N` (common/invariable, translated as `inv`).
4. `neclass/S:person;G:location;O:organization;V:other` - Fifth digit is the semantic class for proper nouns (feature `neclass`), with possible values `S` (translated as `person`), `G` (translated as `location`), `O` (translated as `organization`), and `V` (translated as `other`).
5. `grade/V:evaluative` - sixth digit is the grade (feature `grade`) with possible values `V` (translated as `evaluative`).

If a feature is underspecified or not appliable, a zero (`0`) is expected in the appropriate position of the PoS tag.

The following tag translations would result of the example rule described above:

EAGLES	PoS tag short version	morphological features
NCMS00	NC	pos=noun, type=common, gen=masc, num=sing
NCFC00	NC	pos=noun, type=common, gen=fem, num=common
NCFP00	NC	pos=noun, type=common, gen=fem, num=plur, grade=evaluative
NP0000	NP	pos=noun, type=proper
NP00G0	NP	pos=noun, type=proper, neclass=location

Semantic Database Module

This module is not a main processor in the default analysis chain, but it is used by the other modules that need access to the semantic database: The sense annotator `senses`, the word sense disambiguator `ukb_wrap`, the dependency parser `dep_txala`, and the coreference solver `coref`.

Moreover, this module can be used by the applications to enrich or post process the results of the analysis.

The API for this module is

```
class semanticDB {
  public:
    /// Constructor
    semanticDB(const std::string &);

    /// Compute list of lemma-pos to search in WN for given
word,
    /// according to mapping rules.
    void get_WN_keys(const std::wstring &,
                     const std::wstring &,
                     const std::wstring &,

std::list<std::pair<std::wstring,std::wstring> > &) const;

    /// get list of words for a sense+pos
    std::list<std::string> get_sense_words(const std::string &,
                                           const std::string &)
const;

    /// get list of senses for a lemma+pos
    std::list<std::string> get_word_senses(const std::string &,
                                           const std::string &)
const;

    /// get sense info for a sensecode+pos
    sense_info get_sense_info(const std::string &, const
std::string &) const;
};
```

The constructor receives a configuration file, with the following contents:

- A section `<WNPosMap>` which establishes which PoS found in the morphological dictionary should be mapped to each WN part-of-speech. Rule format is described below
- A section `<DataFiles>` specifying the knowledge bases required by the algorithm. This section may contain up to three keywords, with the format:

```
<DataFiles>
senseDictFile  ./senses30.src
wnFile  ../common/wn30.src
formDictFile  ./dicc.src
</DataFiles>
```

`senseDictFile` is the sense repository, with the format described below.
`wnFile` is a file stating hyperonymy relations and other semantic information for each sense. The format is described below.
`formDictFile` may be needed if mapping rules in `<WNPosMap>` require it. It is a regular form dictionary file with morphological information, as described in section Dictionary Search.

PoS mapping rules

Each line in section `<WNPosMap>` defines a mapping rule, with format `FreeLing-PoS WN-PoS search-key`
where:

- `FreeLing-PoS` is a prefix for a FreeLing PoS tag
- `WN-Pos` must be one of `n`, `a`, `r`, or `v`
- `search-key` defines what should be used as a lemma to search the word in WN files. The given `search-key` may be one of `L`, `F`, or a FreeLing PoS tag. If `L` (`F`) is given, the word lemma (form) will be searched in WN to find candidate senses. If a FreeLing PoS tag is given, the form for that lemma with the given tag will be used.

Example 1: For English, we could have a mapping like:

```
<WNposMap>
N n L
J a L
R r L
V v L
VBG a F
</WNposMap>
```

which states that for words with FreeLing tags starting with `N`, `J`, `R`, and `V`, lemma will be searched in wordnet with PoS `n`, `a`, `r`, and `v` respectively. It also states that words with tag `VBG` (e.g. *boring*) must be searched as adjectives (`a`) using their form (that is,

boring instead of its lemma *bore*). This may be useful, for instance, if FreeLing English dictionary assigns to that form a gerund analysis (`bore VBG`) but not an adjective one, and WordNet contains that word as an adjective.

Example 2: A similar example for Spanish, could be:

```
<WNposMap>
N n L
A a L
R r L
V v L
VMP a VMP00SM
</WNposMap>
```

which states that for words with FreeLing tags starting with `N` , `A` , `R` , and `V` , lemma will be searched in wordnet with PoS `n` , `a` , `r` , and `v` respectively. It also states that words with tag starting with `VMP` (e.g. *cansadas*) must be searched as adjectives (`a`) using the form for the same lema (i.e. *cansar*) that matches the tag `VMP00SM` (resulting in *cansado*). This is useful to have participles searched as adjectives, since FreeLing Spanish dictionary doesn't contain any participle as adjective, but esWN does.

Sense Dictionary File

This source file (e.g. `senses30.src` provided with FreeLing) must contain the word list for each synset, one entry per line. Each line has format:
`sense word1 word2 ...`
E.g. `00045250-n actuation propulsion`
`00050652-v assume don get_into put_on wear`

Sense codes can be anything (assuming your later processes know what to do with them) provided they are unambiguous (there are not two lines with the same sense code). The files provided in FreeLing contain WordNet 3.0 synset codes.

Words in the line can be multiwords, provided they are lowercased and glued toghether with underscores (see example above). Note that a multiword will only be found in the sense dictionary if it has been previously glued toghether by the multiwords module

WordNet file

This source file (e.g. `wn30.src` provided with FreeLing) must contain at each line the information relative to a sense, with the following format:

```
sense hypern:hypern:...:hypern semfile TopOnto:TopOnto:...:TopOnto sumo cyc
```

That is: the first field is the sense code. The following fields are:

- A colon-separated list of hypernym synsets.
- WN semantic file the synset belongs to.
- A colon-separated list of EuroWN TopOntology codes valid for the synset.
- A code for an equivalent (or near) concept in SUMO ontology. See SUMO documentation for a description of the code syntax.
- A code for an equivalent concept in OpenCyC ontology.

Note that the only WN relation encoded here is hypernymy. Note also that semantic codes such as WN semantic file or EWN TopOntology features are simply (lists of) strings. Thus, you can include in this file any ontological or semantic information you need, just substituing the WN-related codes by your own semantic categories.

Approximate search dictionary

This class wraps a `libfoma` FSM and allows fast retrieval of similar words via string edit distance based search.

The API of the class is the following:

```
class foma_FSM {

  public:
    typedef enum {FSM_DICC,FSM_COMPOUNDS} FSM_Type;

    /// build automaton from file
    foma_FSM(const std::wstring &,
             FSM_Type type=FSM_DICC,
             const std::wstring &mcost=L"");
    /// clear
    ~foma_FSM();

    /// Use automata to obtain closest matches to given form,
    /// and add them to given list.
    void get_similar_words(const std::wstring &,
                           std::list<std::pair<std::wstring,int>
> &) const;
    /// set maximum edit distance of desired results
    void set_cutoff_threshold(int);
    /// set maximum number of desired results
    void set_num_matches(int);
    /// Set cost for basic SED operations (insert, delete,
substitute)
    void set_basic_operation_cost(int);
    /// Set cost for a particular SED operation (replace "in"
with "out")
    void set_operation_cost(const std::wstring &in,
                            const std::wstring &out,
                            int cost);
  };
```

The constructor of the module requests one parameter stating the file to load, second optional parameter stating whether the automata must recognize just the words in the given dictionary file, or it must recognize compounds of these words (that is, the language $L(L+)$). The third parameter -also optional- is a file with the cost matrix for SED operations. If the cost matrix is not given, all operations default to a cost of 1 (or to the value set with the method `set_basic_operation_cost`).

The automata file may have extension `.src` or `.bin`. If the extension is `.src`, the file is intepreted as a text file with one word per line. The FSM is built to recognize the vocabulary contained in the file.

If the extension is `.bin`, the file is intepreted as a binary `libfoma` FSM. To compile such a binary file, FOMA command line front-end must be used. The front-end is not included in FreeLing. You will need to install FOMA if you want to create binary FSM files. See FOMA documentation for details.

A cost matrix for SED operations may be specified only for text FSMs (i.e., for `.src` files). To use a cost matrix with a `.bin` file, you need to compile it into the automata using FOMA front-end.

The format of the cost matrix must comply with FOMA formats. See FOMA documentation, or examples provided in `data/common/alternatives` in FreeLing tarball.

The method `get_similar_words` will receive a string and return a list of entries in the FSM vocabulary sorted by string edit distance to the input string.

Feature Extraction Module

Machine Learning based modules (such as BIO named entity recognition or classification modules) require the encoding of each word to classify as a feature vector. The conversion of words in a sentence to feature vectors, is performed by this module. The features are task-oriented, so they vary depending on what is being classified. For this reason, the encoding is not hard-wired in the code, but dinamically performed interpreting a set of feature rules.

Thus, the Feature Extraction Module converts words in a sentence to feature vectors, using a given set of rules.

The API of this module is the following:

```
class fex {
  private:

  public:
    /// constructor, given rule file, lexicon file (may be
empty),
    /// and custom functions
    fex(const std::wstring&, const std::wstring&,
        const std::map<std::wstring,const feature_function *>
&);

    /// encode given sentence in features as feature names.
    void encode_name(const sentence &,
std::vector<std::set<std::wstring> > &);
    /// encode given sentence in features as integer feature
codes
    void encode_int(const sentence &, std::vector<std::set<int>
> &);
    /// encode given sentence in features as integer feature
codes and
    /// as features names
    void encode_all(const sentence &,
std::vector<std::set<std::wstring> > &,
                    std::vector<std::set<int> > &);
```

```
    /// encode given sentence in features as feature names.
    /// Return result suitable for Java/perl APIs
    std::vector<std::list<std::wstring> > encode_name(const
sentence &);
    /// encode given sentence in features as integer feature
codes.
    /// Return result suitable for Java/perl APIs
    std::vector<std::set<int> > encode_int(const sentence &);

    /// clear lexicon
    void clear_lexicon();
    /// encode sentence and add features to current lexicon
    void encode_to_lexicon(const sentence &);
    /// save lexicon to a file, filtering features with low
occurrence rate
    void save_lexicon(const std::wstring &, double) const;
};
```

The class may be used to encode a corpus and generate a feature lexicon, or to encode a corpus filtering the obtained features using a previously generated feature lexicon.

The rules may call custom feature functions, provided the instantianting program provides pointers to call the apropriate code to compute them.

Once the class is instantiated, it can ben used to encode sentences in feature vectors. Features may be obtained as strings (feature names) or integers (feature codes).

The constructor of the class receives a `.rgf` file containing feature extraction rules, a feature lexicon file (mapping feature names to integer codes), and a `map<string,feature_function>` used to define custom feature functions).

If the lexicon file name is empty, features will be assigned an integer code, and the generated lexicon can be saved. This is useful when encoding training corpus and feature codes have not been set yet.

Feature Extraction Rule File

Feature extraction rules are defined in a `.rgf` file. This section describes the format of the file. The syntax of the rules is described further below.

Rules are grouped in packages. Begin and end of a package is marked with the keywords `RULES` and `ENDRULES`. Packages are useful to simplify the rules, and to speed up feature computation avoiding computing the same features several times.

A line with format `TAGSET filename` may precede the rule packages definition. The given `filename` will be interpreted as a relative path (based on the `.rgf` location) to a tagset definition file that will be used to obtain short versions of PoS tags. The `TAGSET` line is needed only if the short tag property `t` is used in some rule.

The `RULES` package starting keyword must be followed by a condition on word properties. Rules in a package will onlly be applied to those words matching the package condition, thus avoiding unnecessary tests.

For instance, the rules in the package:

```
RULES t matches ^NP
 ...
ENDRULES
```

will be applied only for words with a PoS tag (`t`) starting with `NP`. The same result could have been obtained without the package if the same condition was added to each rule, but then, applicability tests for each rule on each word would be needed, resulting in a higher computational cost.

The package condition may be `ALL`. In this case, rules contained in the package will be checked for all words in the sentence. This condition has also an extra effect: the features extracted by rules in this package are cached, in order to avoid repeating computations, e.g. if a rule uses a window to get features from neighbour words.

For instance, the rule:

```
RULES ALL
 punct_mark@   [-2,2]   t matches ^F
ENDRULES
```

will generate, for each word, features indicating which words in the surrounding two words (left and right) are punctuation symbols (`F`).

With this rule applied to the sentence "*Hi ! , said John .*" the word *said* would get the features `punct_mark@-1` (comma to the left of *said*), `punct_mark@-2` (exclamation mark), and `punct_mark@2` (dot after *John*). The word *John* would get the features `punct_mark@-2` and

`punct_mark@1`. Since the package has condition `ALL`, the features are computed once per word, and then reused (that is, the fact that the comma is a punctuation sign will be checked only once, regardless of the size of the sentence and the size of the windows in the rules).

Rule Syntax

Each rule has following syntax:
`feature-name-pattern window condition`

- `feature-name-pattern` is a string that describes what the generated feature name will be. Some special characters allow the insertion of variable values in the feature name. Details on feature patterns are provided below.
- `window` is a range in the format `[num,num]`, and states the words around the target word for which the feature has to be computed. A window of `[0,0]` means that the feaure is only checked for the target word.
- `condition` is the condition that a word has to satisfy in order to get the features extracted by the rule. Details on condition sintax are provided below.

Feature Name Pattern Syntax

Each feature rule has a `feature-name-pattern` that describes how the generated feature name will be.

The following characters are special and are interpreted as variables, and replaced by the corresponding values:

- Character `@`: will be replaced with the relative position of the matching word with respect to the target word. Thus, the rule `punct_mark@ [-2,2] t matches ^F` will generate a different feature for each word in the window that is a punctuation sign (e.g. `punct_mark@-2` and `punct_mark@1` for the word *John* in the above example).

  But the rule `punct_mark [-2,2] t matches ^F` will generate the same feature for all words in the window that are punctuation signs, since position is not encoded in the feature pattern (i.e. it will generate `punct_mark` twice for the word *John* in the above example). Repeated features are stored only once.

- Character `$` introduces a variable that must have the format: `$var(position)`.
  - Allowed variable names are: `W` (word form, in its original casing), `w` (word form, lowercased), `l` (word lemma), `T` (word full PoS tag), `t` (word short PoS tag), `a` (word lemma+Pos tag). All above variables refer to the analysis selected by the

tagger. Variable names may be prefixed with `p` (e.g. `pT`, `pl`, `pa`, etc.) which will generate the feature for all possible analysis of the word, not just the one selected by the tagger.

- The `position` indicates from which word (relative to the focus word) the value for the variable must be taken.

For instance, the pattern: `pbig@:$w(0)_$pt(1)` will extract features that will contain the relative position (`@`), plus a bigram made of the word form of the current word in the window (`$w(0)`) plus each possible short PoS tag of the word right of it (`$pt(1)`).

In the sentence "*John lives here .*", the features for word *here* in a window of [-2,0] with the above pattern would be: `pbig@-2:john_VBZ` (word form for *John* plus first possible tag for *lives*), `pbig@-2:john_NNS` (word form for *John* plus second possible tag for *lives*), `pbig@-1:lives_RB` (word form for *lives* plus first possible tag for *here*), and `pbig@0:here_Fp` (word form for *here* plus first tag for the dot). Note that there are two features generated for window position -2 because the word *lives* has two possible PoS tags.

- Curly brackets `{ }` have two possible interpretations, depending on what they contain:
  1. If the brackets enclose a regex match variable (e.g $0, $1, $2, ...), then they are replaced with the string matching the corresponding (sub)expression. This only makes sense if the condition of the rule included a regular expression match. If it is not the case, results are undefined (probably a segmentation violation).
  2. If the brackets do not contain a regex match variable, then the content is interpreted as call to a custom feature function. It must have the format `{functname(position)}`, where `functname` is the name of the function as declared in the custom feature functions map (see below), and the `position` parameter is the relative position to the target word, and is interpreted in the same way than in the primitive features `$w(position)`, `$t(position)`, etc., described above.
     E.g., the pattern: `{quoted(-1)}_{quoted(0)}` would generate a feature similar to that of the pattern: `t(-1)_t(0)` but using the result of the custom function `quoted` instead of the PoS tag for the corresponding word.

Feature Rules Condition Syntax

Conditions control the applicability of a rule or a rule package to a certain word.

A condition may be `ALL` which is satisfied by any word. A condition may be simple, or compund of several conditions, combined with the logical operadors `AND` and `OR`. The operators in a condition must be homogeneous (i.e. either all of them `AND` or all of them

`OR`), mixed conditions are not allowed (note that an `OR` condition is equivalent to writing two rules that only differ on the condition).

Single conditions consist of a word property, an operation, and an argument. Available word properties are:

- `W`: Word form, original casing.
- `w`: Word form, lowercased.
- `l`: Lemma of the analysis selected by the tagger.
- `t`: PoS tag (short version) of the analysis selected by the tagger.
- `T`: PoS tag (full version) of the analysis selected by the tagger.
- `pl`: List of all possible lemmas for the word.
- `pt`: List of all possible short PoS tags for the word.
- `pT`: List of all possible full PoS tags for the word.
- `na`: Number of analysis of the word.
- `u.` *i*: *i*-th element of the word `user` field (see description of `word` class.)

Note that all word properties (including `na`) are either strings or lists of strings.

The available primitive operations to build single conditions are the following:

1. `<property> is <string>`: String identity.
2. `<property> matches <regexp>`: Regex match. If the regex is parenthesized, (sub)expression matches `$0`, `$1`, `$2`, etc. are stored and can be used in the feature name pattern.
3. `<property-list> any_in_set <filename>` (or simply `in_set`): True iff any property in the list is found in the given file.
4. `<property-list> all_in_set <filename>`: True iff all properties in the list are found in the given file.
5. `<property-list> some_in_set <filename>`: True iff at least two properties in the list are found in the given file.

Operators can be negated with the character `!`. E.g. `!is`, `!matches`, etc.

For file operators expecting lists, the property may be a single string (list of one element).

Some sample valid conditions:

- `t is NC` true if the short version of the tag equals `NC`.
- `T matches ^NC.S..` true if the long version of the tag matches the given regular expression.
- `pl in_set my/data/files/goodlemmas.dat` rue if any possible lemma for the word is found in the given file.
- `l !in_set my/data/files/badlemmas.dat` true if selected lemma for the word is not found

in the given file.

- `w matches ...$` Always true. Will set the match variable `$0` to the last three characters of the word, so it can be used in the feature name pattern (e.g. to generate a feature with the suffix of each word).

Adding custom feature functions

Custom feature functions can be defined, and called from the `.rgf` file enclosed in curly brackets (e.g.: `{quoted(0)}`). Calls to custom feature functions in the `.rgf` file must have one integer parameter, indicating a word position relative to the target word.

Actual code computing custom feature functions must be provided by the caller. the constructor accepts a parameter containing a map `std::map<std::wstring,const feature_function*>` , associating the custom function name used in the rule file with a `feature_function` pointer.

Custom feature functions must be classes derived from class `feature_function` :

```
class feature_function {
    public:
      virtual void extract (const sentence &s,
                            int pos,
                            std::list<std::wstring> &) const=0;
      /// Destructor
      virtual ~feature_function() {};
  };
```

They must implement a method `extract` that receives the sentence, the position of the target word, and a list of strings where the resulting feature name (or names if more than one is to be generated) will be added.

For instance, the example below generates the feature name `in_quotes` when the target word is surrounded by words with the `Fe` PoS tag (which is assigned to any quote symbol by the punctuation module).

```
class fquoted : public feature_function {
    public:
      void extract (const sentence &sent, int i,
std::list<std::wstring> &res) const {
        if ( (i>0 and sent[i-1].get_tag()==L"Fe") and
             (i<(int)sent.size()-1 and
sent[i+1].get_tag()==L"Fe") )
          res.push_back(L"in_quotes");
      }
  };
```

We can associate this function with the function name `quoted` adding the pair to a map:

```
map<wstring,const feature_function*> myfunctions;
myfunctions.insert(make_pair(L"quoted", (feature_function *) new
fquoted()));
```

If we now create a `fex` object passing this map to the constructor, the created instance will call `fquoted::extract` with the appropriate parameters whenever `quoted` feature is used in a rule in the `.rgf` file.
For instance, we could create the rule:
`{quoted(0)}@_t(1) [-5,5] ALL`
which will generate a feature with the position and the tag of the word after the quote for any quote in the window `[-5,5]` around the target word (e.g. `in_quotes@-2_NC`, `in_quotes@3_DT`, etc.)

Note that there are three naming levels for custom feature functions:

- The name of the feature itself, which will be generated by the extractor and will appear in the feature vectors (`in_quotes` in the above example).
- The name of the function that will be called from the extraction rules in the `.rgf` file (`quoted` in the above example).
- The name of the class derived from `feature_function` that has a method `extract` which actually computes the feature (`fquoted` in the above example).

Input/Output handling modules

FreeLing package includes a few classes that can convert between FreeLing analyzed documents and an output representantion. New classes can be created by the user to meet specific needs.

The I/O handlers are derived from two top abstract classes: `input_handler` and `output_handler`.

Derived classes handle a particular I/O format, and must implement the appropriate methods.

```
class input_handler {

 public:
   /// constructor.
   input_handler ();
   /// destructor
   ~input_handler();

   /// load partially analyzed sentences from 'lines' into a
list of sentences
   virtual void input_sentences(const std::wstring &lines,
                                std::list<freeling::sentence>
&ls) const = 0;
   /// load partially analyzed sentences from 'lines' into a
document
   virtual void input_document(const std::wstring &lines,
                               freeling::document &doc) const;
};
```

```
class output_handler {

 public:
   /// empty constructor
   output_handler();
   /// destructor
   ~output_handler();

   /// load tagset rules for PoS shortening and MSD descriptions
   void load_tagset(const std::wstring &ftag);
   /// set language
   void set_language(const std::wstring &lg);

   /// Print appropriate header for the output
   /// format (e.g. XML header or tag opening)
   virtual void PrintHeader(std::wostream &sout) const;
   /// print appropriate footer (e.g. close XML tags)
   virtual void PrintFooter(std::wostream &sout) const;
   /// print given sentences to sout in appropriate format (no
headers)
   virtual void PrintResults(std::wostream &sout,
                             const std::list<freeling::sentence>
&ls) const=0;
   virtual std::wstring PrintResults(const
std::list<freeling::sentence> &ls) const;

   /// print given document to sout in appropriate format,
including headers.
   virtual void PrintResults(std::wostream &sout,
                             const freeling::document &doc)
const = 0;
   virtual std::wstring PrintResults (const freeling::document
&doc) const;
};
```

Output handlers

Currently, the following output handlers are implemented. All of them are derived from `output_handler`.

FreeLing Output

The module `output_freeling` produces the classical FreeLing output (column-like for lower analysis levels, and parenthesized trees for higher levels). The desired output can be configured by the calling application activating or deactivating print levels.

```
class output_freeling : public output_handler {
 public:
   // constructor.
   output_freeling ();
   output_freeling(const std::wstring &cfgFile);
   ~output_freeling ();

   /// output a parse tree
   void PrintTree (std::wostream &sout,
                   freeling::parse_tree::const_iterator n,
                   int depth) const;
   /// output a dependency tree
   void PrintDepTree (std::wostream &sout,
                      freeling::dep_tree::const_iterator n,
                      int depth) const;
   /// output a predicate structure
   void PrintPredArgs (std::wostream &sout,
                       const freeling::sentence &s) const;
   /// output a word and its morphological information
   void PrintWord (std::wostream &sout,
                   const freeling::word &w,
                   bool only_sel=true,
                   bool probs=true) const;
   /// Output coreference groups
   void PrintCorefs(std::wostream &sout,
                    const freeling::document &doc) const;
   /// Output semantic Graph
   void PrintSemgraph(std::wostream &sout,
                      const freeling::document &doc) const;

   /// print given sentences to sout in appropriate format
```

```
    void PrintResults (std::wostream &sout,
                       const std::list<freeling::sentence> &ls)
const;
    /// print given document to sout in appropriate format
    void PrintResults(std::wostream &sout,
                      const freeling::document &doc) const;

    /// inherit other methods from abstract class
    using output_handler::PrintResults;

    // activate/deactivate printing levels
    void output_senses(bool);
    void output_all_senses(bool);
    void output_phonetics(bool);
    void output_dep_tree(bool);
    void output_corefs(bool);
    void output_semgraph(bool);
};
```

CoNLL Output

The module `output_conll` produces a CoNLL-like column format. The default format is not exactly any of the CoNLL competitions, but an adapted version.

The module will always print all information avaliable in the document (i.e. if the document is parsed, a column for the parse tree will be generated).

The constructor may be called with a configuration file stating which columns should be printed and in which order. The configuration file must contain a section `<Type>` containing just the keyword `conll` (just to explicit that this file is for this kind of output_handler), as well as a section `<Columns>` specifying the list of columns that must appear in the output, and in which order. The file may also contain a section `<TagsetFile>` pointing which tagset definition file should be used (this section is only required if `SHORT_TAG` or `MSD` are specified in the `<Columns>` section.

The default behaviour of this module (that is, if no configuration file is provided), is the same than with the configuration file:

```
<Type>
conll
</Type>
<TagsetFile>
./tagset.dat
</TagsetFile>
<Columns>
ID FORM LEMMA TAG SHORT_TAG MSD NEC SENSE SYNTAX DEPHEAD DEPREL
COREF SRL
</Columns>
```

Valid column names are:

- `ID` : token id (position of word in the sentence)
- `SPAN_BEGIN` : character position of the word in the original text.
- `SPAN_END` : character position of the end of word in the original text.
- `FORM` : word form
- `LEMMA` : word lemma
- `TAG` : Complete PoS tag
- `SHORT_TAG` : Short Pos tag (as shortened by the Tagset definition file)
- `MSD` : Morphosyntactic features (as produced by the Tagset definition file)
- `NEC` : Named Entity Classifier output in B-I-O format
- `SENSE` : Sense selected by the used WSD module (or first sense if no disambiguation was used)
- `ALL_SENSES` : List of all possible senses for the word
- `SYNTAX` : Information about constituents opening/closing in this word
- `DEPHEAD` : Head of the word in the dependency tree
- `DEPREL` : Syntactic function of the word with regard to its head.
- `COREF` : Coreference groups opening/closing in this word
- `SRL` : Semantic Role Labelling predicates and arguments in CoNLL format. This may take more than one column, and must always be the last field in the list.
- `USERx` : (where `x` is a number) Content of the `x` -th position in the `user` vector of each word (this vector is ignored by FreeLing, it is intended for the user application to store any needed information).

```
class output_conll : public output_handler {

 public:
   /// empty constructor.
   output_conll ();
   /// constructor from cfg file
   output_conll (const std::wstring &cfgFile);
   /// destructor.
   ~output_conll ();

   /// Fill conll_sentence from freeling::sentence
   void freeling2conll(const freeling::sentence &s,
                       conll_sentence &cs) const;

   /// print given sentences to sout in appropriate format
   void PrintResults (std::wostream &sout,
                      const std::list<freeling::sentence> &ls)
const;
   /// print given a document to sout in appropriate format
   void PrintResults(std::wostream &sout,
                     const freeling::document &doc) const;
   /// inherit other methods
   using output_handler::PrintResults;
};
```

XML Output

The module `output_xml` produces an XML representation of the analyzed document.

The module will always print all information avaliable in the document (i.e. if the document is parsed, an XML tag for the parse tree will be generated).

```
class output_xml : public output_handler {

 public:
   /// constructor.
   output_xml ();
   output_xml (const std::wstring &cfgFile);
   /// destructor.
   ~output_xml ();

   /// print XML file header
   void PrintHeader(std::wostream &sout) const;
   /// print XML file footer
   void PrintFooter(std::wostream &sout) const;

   /// print given sentences to sout in appropriate format
   void PrintResults (std::wostream &sout,
                      const std::list<freeling::sentence> &ls)
const;
   /// print given a document to sout in appropriate format
   void PrintResults(std::wostream &sout,
                     const freeling::document &doc) const;
   /// inherit other methods
   using output_handler::PrintResults;
};
```

The Schema for the XML generated by this module is:

```
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">


  <!-- ********************************** -->
  <!-- Type definition for parse tree nodes -->
  <!-- ********************************** -->
  <xs:complexType name="nodeType">
    <!-- Each node can have list of child nodes -->
    <xs:sequence>
      <xs:element name="node" type="nodeType" minOccurs="0"
```

```
maxOccurs="unbounded"/>
    </xs:sequence>
    <!-- Content of the tree nodes -->
    <xs:attribute type="xs:boolean" name="leaf" use="optional"/>
    <xs:attribute type="xs:boolean" name="head" use="optional"/>
    <xs:attribute type="xs:string" name="label" use="optional"/>
    <xs:attribute type="xs:string" name="token" use="optional"/>
    <xs:attribute type="xs:string" name="word" use="optional"/>
  </xs:complexType>

  <!-- ********************************************* -->
  <!-- Type definition for dependency tree nodes -->
  <!-- ********************************************* -->
  <xs:complexType name="depnodeType">
    <!-- Each node can have list of child nodes -->
    <xs:sequence>
      <xs:element name="depnode" type="depnodeType"
minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <!-- Content of the dep tree nodes -->
    <xs:attribute type="xs:string" name="token" use="optional"/>
    <xs:attribute type="xs:string" name="function"
use="optional"/>
    <xs:attribute type="xs:string" name="word" use="optional"/>
  </xs:complexType>


  <!-- ********************************** -->
  <!-- **         MAIN DOCUMENT          ** -->
  <!-- ********************************** -->
  <xs:element name="document">
    <xs:complexType>

      <!-- A document contains paragraphs, coreferences, and
semantic_graph -->
      <xs:sequence>

      <!-- paragraphs -->
      <xs:sequence>
        <xs:element name="paragraph" maxOccurs="unbounded"
```

```
minOccurs="1">
          <xs:complexType>

            <!-- A paragraph contains a list of sentences -->
            <xs:sequence>
              <xs:element name="sentence" maxOccurs="unbounded"
minOccurs="1">
                <xs:complexType>

                  <!-- a sentence contains tokens, parse_tree,
dep_tree, and predicates -->
                  <xs:sequence>

                  <!-- A sentence contains a list of tokens -->
                  <xs:sequence>
                    <xs:element name="token"
maxOccurs="unbounded" minOccurs="1">
                      <xs:complexType>
                         <xs:sequence>
                           <xs:element name="analysis"
maxOccurs="unbounded" minOccurs="0">
                             <xs:complexType>
                               <xs:attribute type="xs:string"
name="lemma" use="optional"/>
                               <xs:attribute type="xs:string"
name="tag" use="optional"/>
                               <xs:attribute type="xs:string"
name="ctag" use="optional"/>
                               <xs:attribute type="xs:string"
name="pos" use="optional"/>
                               <xs:attribute type="xs:string"
name="vform" use="optional"/>
                               <xs:attribute type="xs:string"
name="wn" use="optional"/>
                               <xs:attribute type="xs:string"
name="type" use="optional"/>
                               <xs:attribute type="xs:string"
name="punctenclose" use="optional"/>
                               <xs:attribute type="xs:string"
name="num" use="optional"/>
```

```
                              <xs:attribute type="xs:string"
name="person" use="optional"/>
                              <xs:attribute type="xs:string"
name="neclass" use="optional"/>
                              <xs:attribute type="xs:string"
name="nec" use="optional"/>
                              <xs:attribute type="xs:double"
name="prob" use="optional"/>
                            </xs:complexType>
                            </xs:element>
                          </xs:sequence>
                        <xs:attribute type="xs:string" name="id"
use="required"/>
                        <xs:attribute type="xs:string"
name="form" use="required"/>
                        <xs:attribute type="xs:string"
name="phon" use="optional"/>
                        <xs:attribute type="xs:string"
name="lemma" use="optional"/>
                        <xs:attribute type="xs:string"
name="tag" use="optional"/>
                        <xs:attribute type="xs:string"
name="ctag" use="optional"/>
                        <xs:attribute type="xs:string"
name="pos" use="optional"/>
                        <xs:attribute type="xs:string"
name="vform" use="optional"/>
                        <xs:attribute type="xs:string" name="wn"
use="optional"/>
                        <xs:attribute type="xs:string"
name="type" use="optional"/>
                        <xs:attribute type="xs:string"
name="punctenclose" use="optional"/>
                        <xs:attribute type="xs:string"
name="num" use="optional"/>
                        <xs:attribute type="xs:string"
name="person" use="optional"/>
                        <xs:attribute type="xs:string"
name="neclass" use="optional"/>
                        <xs:attribute type="xs:string"
```

```
name="nec" use="optional"/>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>

                <!-- A sentence may contain a parse_tree -->
                <xs:sequence>
                  <xs:element name="constituents"
minOccurs="0" maxOccurs="1">
                    <xs:complexType>
                      <!-- The tree contains just a root node
-->
                      <xs:all>
                        <xs:element name="node"
type="nodeType" />
                      </xs:all>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>

                <!-- A sentence may contain a dependency tree
-->
                <xs:sequence>
                  <xs:element name="dependencies"
minOccurs="0" maxOccurs="1">
                    <xs:complexType>
                      <!-- The tree contains just a root node
-->
                      <xs:all>
                        <xs:element name="depnode"
type="depnodeType" />
                      </xs:all>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>

                <!-- A sentence may contain a list of
predicates -->
                <xs:sequence>
                  <xs:element name="predicates" maxOccurs="1"
```

```
minOccurs="0">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="predicate"
maxOccurs="unbounded" minOccurs="1">
                            <xs:complexType mixed="true">

                              <!-- A predicate may contain a
list of arguments -->
                              <xs:sequence>
                                <xs:element name="argument"
maxOccurs="unbounded" minOccurs="0">
                                  <xs:complexType>
                                    <!-- Argument information --
>
                                    <xs:simpleContent>
                                      <xs:extension
base="xs:string">
                                        <xs:attribute
type="xs:string" name="role" use="optional"/>
                                        <xs:attribute
type="xs:string" name="words" use="optional"/>
                                        <xs:attribute
type="xs:string" name="head_token" use="optional"/>
                                        <xs:attribute
type="xs:string" name="from" use="optional"/>
                                        <xs:attribute
type="xs:string" name="to" use="optional"/>
                                      </xs:extension>
                                    </xs:simpleContent>
                                  </xs:complexType>
                                </xs:element>
                              </xs:sequence>

                              <!-- Predicate information -->
                              <xs:attribute type="xs:string"
name="id" use="optional"/>
                              <xs:attribute type="xs:string"
name="head_token" use="optional"/>
                              <xs:attribute type="xs:string"
```

```
name="sense" use="optional"/>
                              <xs:attribute type="xs:string"
name="words" use="optional"/>

                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                    <!-- End of 'predicates' element -->

                  </xs:sequence>
                  <!-- End of 'predicates' list -->

                  </xs:sequence>
                  <!-- End of sentence subelements -->

                  <!-- 'sentence' element attributes -->
                  <xs:attribute type="xs:string" name="id"
use="required"/>

                </xs:complexType>
              </xs:element>
              <!-- End of 'sentence' element -->

            </xs:sequence>
            <!-- End of list of sentences in paragraph -->

          </xs:complexType>
        </xs:element>
        <!-- End of 'paragraph' element -->

      </xs:sequence>
      <!-- End of list of paragraphs in document -->

      <!-- A document may contain a list of coreference sets -->
      <xs:sequence>
        <xs:element name="coreferences" maxOccurs="1"
minOccurs="0">
          <xs:complexType>
```

```
            <xs:sequence>
              <xs:element name="coref" maxOccurs="unbounded"
minOccurs="1">
                <xs:complexType>
                 <!-- A coreference group contains a list of
mentions -->
                  <xs:sequence>
                    <xs:element name="mention"
maxOccurs="unbounded" minOccurs="1">
                      <xs:complexType>
                            <xs:attribute type="xs:string"
name="id" use="required"/>
                            <xs:attribute type="xs:string"
name="from" use="required"/>
                            <xs:attribute type="xs:string"
name="to" use="required"/>
                            <xs:attribute type="xs:string"
name="words" use="required"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <!-- Coreference group information -->
                  <xs:attribute type="xs:string" name="id"
use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <!-- End of 'coreferece' element in document -->

      <!-- A document may contain a semantic graph -->
      <xs:sequence>
        <xs:element name="semantic_graph" maxOccurs="1"
minOccurs="0">
          <xs:complexType>

            <!-- A semantic graph contains a list of 'entities'
and 'frames' -->
```

```
            <xs:sequence>

              <!-- 'entity' elements in the graph -->
              <xs:element name="entity" maxOccurs="unbounded"
minOccurs="0">
                <xs:complexType>

                  <xs:sequence>
                  <!-- Entity contains a list of mentions -->
                  <xs:sequence>
                    <xs:element name="mention"
maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:simpleContent>
                          <!-- Basic mention information -->
                          <xs:extension base="xs:string">
                            <xs:attribute type="xs:string"
name="id" use="optional"/>
                            <xs:attribute type="xs:string"
name="words" use="optional"/>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>

                  <!-- Entity may contains a list of synonyms --
>
                  <xs:sequence>
                    <xs:element name="synonym"
maxOccurs="unbounded" minOccurs="0">
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:extension base="xs:string">
                            <xs:attribute type="xs:string"
name="lemma" use="optional"/>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
```

```
                </xs:sequence>

                <!-- Entity may contains a list of URI -->
                <xs:sequence>
                  <xs:element name="URI" maxOccurs="unbounded"
minOccurs="0">
                    <xs:complexType>
                      <xs:simpleContent>
                        <xs:extension base="xs:string">
                          <xs:attribute type="xs:string"
name="knowledgeBase" use="optional"/>
                          <xs:attribute type="xs:anyURI"
name="URI" use="optional"/>
                        </xs:extension>
                      </xs:simpleContent>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
                </xs:sequence>

                <!-- Basic entity information -->
                <xs:attribute type="xs:string" name="id"
use="required"/>
                <xs:attribute type="xs:string" name="lemma"
use="required"/>
                <xs:attribute type="xs:string" name="class"
use="optional"/>
                <xs:attribute type="xs:string" name="sense"
use="optional"/>

              </xs:complexType>
            </xs:element>
            <!-- End of 'entity' element -->

            <!-- 'frame' elements in the graph -->
            <xs:element name="frame" maxOccurs="unbounded"
minOccurs="0">
              <xs:complexType>

                <xs:sequence>
```

```
                <!-- Frame may contain a list of arguments -->
                <xs:sequence>
                  <xs:element name="argument"
maxOccurs="unbounded" minOccurs="0">
                    <xs:complexType>
                      <xs:simpleContent>
                        <xs:extension base="xs:string">
                          <xs:attribute type="xs:string"
name="role" use="optional"/>
                          <xs:attribute type="xs:string"
name="entity" use="optional"/>
                        </xs:extension>
                      </xs:simpleContent>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>

                <!-- Frame may contains a list of synonyms -->
                <xs:sequence>
                  <xs:element name="synonym"
maxOccurs="unbounded" minOccurs="0">
                    <xs:complexType>
                      <xs:simpleContent>
                        <xs:extension base="xs:string">
                          <xs:attribute type="xs:string"
name="lemma" use="optional"/>
                        </xs:extension>
                      </xs:simpleContent>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>

                <!-- Frame may contains a list of URI -->
                <xs:sequence>
                  <xs:element name="URI" maxOccurs="unbounded"
minOccurs="0">
                    <xs:complexType>
                      <xs:simpleContent>
                        <xs:extension base="xs:string">
                          <xs:attribute type="xs:string"
```

```
name="knowledgeBase" use="optional"/>
                            <xs:attribute type="xs:anyURI"
name="URI" use="optional"/>
                          </xs:extension>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  </xs:sequence>

                  <!-- Basic frame information -->
                  <xs:attribute type="xs:string" name="id"
use="required"/>
                  <xs:attribute type="xs:string" name="token"
use="required"/>
                  <xs:attribute type="xs:string" name="lemma"
use="required"/>
                  <xs:attribute type="xs:string" name="sense"
use="optional"/>

                </xs:complexType>
              </xs:element>
              <!-- End of 'Frame' element in semantic graph -->

            </xs:sequence>
            <!-- End of list of frames and entities in semantic
graph -->

        </xs:complexType>
      </xs:element>
      </xs:sequence>
      <!-- End of semantic graph element -->

      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!-- End of 'document' element -->

</xs:schema>
```

JSON Output

The module `output_json` produces a JSON object for the analyzed document.

The module will always print all information avaliable in the document (i.e. if the document is parsed, an JSON object for the parse tree will be included in the document object).

```
class output_json : public output_handler {

 public:
   // constructor.
   output_json ();
   output_json (const std::wstring &cfgFile);
   // destructor.
   ~output_json ();

   // print given sentences to sout in appropriate format
   void PrintResults (std::wostream &sout,
                      const std::list<freeling::sentence> &ls)
const;
   // print given a document to sout in appropriate format
   void PrintResults(std::wostream &sout,
                     const freeling::document &doc) const;
   /// inherit other methods
   using output_handler::PrintResults;
};
```

The schema for the output produced by this module is:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "FreeLing Analyzed Document",

  "type": "object",
  "properties": {
    "paragraphs": {
      "type": "array",
      "items": {
        "id": "paragraph",
        "type": "object",
```

```
        "properties": {
          "sentences": {
            "type": "array",
            "items": [
              {
                "id": "sentence",
                "type": "object",
                "properties": {
                  "id": { "type": "string" },
                  "tokens": {
                    "id": "tokens",
                    "type": "array",
                    "items": [
                      {
                        "type": "object",
                        "properties": {
                          "id": { "type": "string" },
                          "form": {"type": "string" },
                          "lemma": {"type": "string" },
                          "tag": { "type": "string" },
                          "ctag": { "type": "string" },
                          "pos": { "type": "string" },
                          "type": { "type": "string" },
                          "gen": { "type": "string" },
                          "num": { "type": "string" },
                          "wn": { "type": "string" },
                          "neclass": { "type": "string" },
                          "nec": { "type": "string" },
                          "mood": { "type": "string" },
                          "tense": { "type": "string" },
                          "person": { "type": "string" }
                        }
                      }
                    ]
                  },
                  "constituents": {
                    "id": "constituents",
                    "type": "array",
                    "items": {
                      "type": "object",
```

```
                    "properties": {
                      "label": { "type": "string" },
                      "head": { "type": "string" },
                      "children": {
                    "type" : "object",
                        "$ref" : "constituents"
                      }
                    }
                  }
                },
                "dependencies": {
                  "id": "dependencies",
                  "type": "object",
                  "properties": {
                    "token": { "type": "string" },
                    "function": { "type": "string" },
                    "word": { "type": "string" },
                    "children": {
                      "type": "array",
                      "items": [
                         {
                           "type": "object",
                       "$ref": "dependencies"
                         }
                      ]
                    }
                  }
                },
                "predicates": {
                  "type": "array",
                  "items": [
                    {
                      "id": "predicate",
                      "type": "object",
                      "properties": {
                        "id": { "type": "string" },
                        "head_token": { "type": "string" },
                        "sense": { "type": "string" },
                        "words": { "type": "string" },
                        "arguments": {
```

```
                            "type": "array",
                            "items": [
                              {
                                "type": "object",
                                "properties": {
                                  "role": { "type": "string" },
                                  "words": { "type": "string" },
                                  "head_token": { "type":
"string" },
                                  "from": { "type": "string" },
                                  "to": { "type": "string" }
                                }
                              }
                            ]
                          }
                        }
                      }
                    ]
                  }
                },
                "required": [
                  "id",
                  "tokens"
                ]
              }
            ]
          }
        },
        "required": [
          "sentences"
        ]
      }
    },
    "coreferences": {
      "id": "coreferences",
      "type": "array",
      "items": [
        {
          "type": "object",
          "properties": {
```

```
            "id": { "type": "string" },
            "mentions": {
              "type": "array",
              "items": [
                {
                  "type": "object",
                  "properties": {
                    "id": { "type": "string" },
                    "from": { "type": "string" },
                    "to": { "type": "string" },
                    "words": { "type": "string" }
                 }
                }
              ]
            }
          }
        }
      ]
    },
    "semantic_graph": {
      "id": "semantic_graph",
      "type": "array",
      "items": [
        {
          "id": "entity",
          "type": "object",
          "properties": {
            "id": {  "type": "string" },
            "lemma": { "type": "string" },
            "class": { "type": "string" },
            "sense": { "type": "string" },
            "mentions": {
              "type": "array",
              "items": [
                {
                  "type": "object",
                  "properties": {
                    "id": { "type": "string" },
                    "from": { "type": "string" },
                    "to": { "type": "string" },
```

```
                    "words": { "type": "string" }
                 }
               }
             ]
           },
           "synonyms": {
             "type": "array",
             "items": [
               { "type": "string" }
             ]
           },
           "URIs": {
             "type": "array",
             "items": [
               {
                 "type": "object",
                 "properties": {
                   "knowledgeBase": { "type": "string" },
                   "URI": { "type": "string" }
                 }
               }
             ]
           }
         }
       },
       {
         "id": "frame",
         "type": "object",
         "properties": {
           "id": { "type": "string" },
           "token": { "type": "string" },
           "lemma": { "type": "string" },
           "sense": { "type": "string" },
           "arguments": {
             "type": "array",
             "items": [
               {
                 "type": "object",
                 "properties": {
                   "role": { "type": "string" },
```

```
                    "entity": { "type": "string" }
                  }
                }
              ]
            },
            "synonyms": {
              "type": "array",
              "items": [
                { "type": "string" }
              ]
            },
            "URIs": {
               "type": "array",
               "items": [
                {
                  "type": "object",
                  "properties": {
                    "knowledgeBase": { "type": "string" },
                    "URI": { "type": "string" }
                  }
                 }
               ]
            }
          }
        }
      ]
    }
  },
  "required": [
    "paragraphs",
  ]
}
```

NAF Output

The module `output_naf` produces a NAF [FSB+14] representation for the analyzed document.

The desired output can be configured by the calling application activating or deactivating annotation layers.

```
class output_naf : public output_handler {
 public:
   // constructor.
   output_naf ();
   output_naf (const std::wstring &cfgFile);
   // destructor.
   ~output_naf ();

   // print given sentences to sout in appropriate format
   void PrintResults (std::wostream &sout,
                      const std::list<freeling::sentence> &ls)
const;
   // print given a document to sout in appropriate format
   void PrintResults(std::wostream &sout,
                     const freeling::document &doc) const;
   /// inherit other methods
   using output_handler::PrintResults;

   // print NAF header
   void PrintHeader(std::wostream &sout) const;
   // print NAF footer
   void PrintFooter(std::wostream &sout) const;

   // activate/deactivate layer for next printings
   void ActivateLayer(const std::wstring &ly, bool b);
};
```

The module `output_train` produces a FreeLing specific format suitable to be used to retrain the PoS tagger (tipically after hand correction of the tagger errors).

```
class output_train : public output_handler {

 public:
   // constructor.
   output_train ();
   ~output_train ();

   void PrintWord (std::wostream &sout,
                   const freeling::word &w,
                   bool only_sel=true,
                   bool probs=true) const;
   // print given sentences to sout in appropriate format
   void PrintResults (std::wostream &sout,
                      const std::list<freeling::sentence> &ls)
const;
   // print given a document to sout in appropriate format
   void PrintResults(std::wostream &sout,
                     const freeling::document &doc) const;
   /// inherit other methods
   using output_handler::PrintResults;
};
```

Input handlers

Currently, the following output handlers are implemented. All of them are derived from `input_handler`.

FreeLing Input

This class will load output produced by FreeLing with `--outlv morfo` or `--outlv tagged`.

```
class input_freeling : public input_handler {
 public:
  // constructor.
  input_freeling ();
  // destructor.
  ~input_freeling ();

  /// load partially analyzed sentences from 'lines' into a list
of sentences
  virtual void input_sentences(const std::wstring &lines,
                               std::list<freeling::sentence>
&ls) const;

  /// load partially analyzed sentences from 'lines' into a
document
  virtual void input_document(const std::wstring &lines,
                               freeling::document &doc) const;

};
```

CoNLL Input

This class will load output produced by `output_conll` (see above).

A configuration file can be provided to the constructor. The configuration file format is the same than for `output_conll` .

```
class input_conll : public input_handler, public conll_handler {

 public:
  // default constructor.
  input_conll();
  // constructor from config file
  input_conll(const std::wstring &fcfg);
  // destructor.
  ~input_conll();

  void input_sentences(const std::wstring &lines,
std::list<freeling::sentence> &ls) const;
  void input_document(const std::wstring &lines,
freeling::document &doc) const;
};
```

The default behaviour of this module (that is, if no configuration file is provided), is the same than with the configuration file:

```
<TagsetFile>
./tagset.dat
</TagsetFile>
<Columns>
ID FORM LEMMA TAG SHORT_TAG MSD NEC SENSE SYNTAX DEPHEAD DEPREL
COREF SRL
</Columns>
```

Using the library from your own application

The library may be used to develop your own NLP application (e.g. a machine translation system, an intelligent indexation module for a search engine, etc.)

To achieve this goal you have to link your application to the library, and access it via the provided API. Since the library is C++, using C++ in your program provides full access to all library functionalities. However, quite complete APIs are provided for Java, Perl, Python, PHP, and Ruby.

Basic Classes

This section briefs the basic C++ classes any application needs to know. For detailed API definition, consult the technical documentation in `doc/html` and `doc/latex` directories, or chapters about Linguistic Data Classes or Language Processing Modules

Linguistic Data Classes

The different processing modules work on objects containing linguistic data (such as a word, a PoS tag, a sentence...).

Your application must be aware of those classes in order to be able to provide to each processing module the right data, and to correctly interpret the module results.

The linguistic classes are:

- `analysis`: A tuple `<lemma, PoS tag, probability, sense list>`
- `word`: A word form with a list of possible analysis.
- `sentence`: A list of words known to be a complete sentence. A sentence may have associated a `parse_tree` object and a `dependency_tree`.
- `parse_tree`: An n-ary tree where each node contains either a non-terminal label, or -if the node is a leaf- a pointer to the appropriate `word` object in the sentence the tree belongs to.
- `dep_tree`: An n-ary tree where each node contains a reference to a node in a `parse_tree`. The structure of the `dep_tree` establishes syntactic dependency relationships between sentence constituents.
- `paragraph`: A list of `sentence` objects that form a paragraph.

- `document`: A list of `paragraph` objects that form the document. A document may have associated a set of coreference groups and a semantic graph.
- `mention`: A mention of an entity in the document. A group of mentions referring to the same entity form a coreference group.
- `semantic_graph`: An entity-relationship graph describing the main actions reported in the text and the principal actors involved in each.

Processing modules

The main processing classes in the library are:

- `tokenizer`: Receives plain text and returns a list of `word` objects.
- `splitter`: Receives a list of `word` objects and returns a list of `sentence` objects.
- `maco`: Receives a list of `sentence` objects and morphologically annotates each `word` object in the given sentences. Includes specific submodules (e.g, detection of date, number, multiwords, etc.) which can be activated at will.
- `tagger`: Receives a list of `sentence` objects and disambiguates the PoS of each `word` object in the given sentences.
- `nec`: Receives a list of `sentence` objects and modifies the tag for detected proper nouns to specify their class (e.g. person, place, organitzation, others).
- `ukb`: Receives a list of `sentence` objects enriches the words with a ranked list of WordNet senses.
- `parser`: Receives a list of `sentence` objects and associates to each of them a `parse_tree` object.
- `dependency`: Receives a list of parsed `sentence` objects and associates to each of them a `dep_tree` object.
- `coref`: Receives a document (containing a list of parsed `sentence` objects) and labels each noun phrase as belonging to a coreference group, if appropriate.

You may create as many instances of each as you need. Constructors for each of them receive the appropriate options (e.g. the name of a dictionary, hmm, or grammar file), so you can create each instance with the required capabilities (for instance, a tagger for English and another for Spanish).

Sample programs

The directory `src/main/simple_examples` in the tarball contains some example programs to illustrate how to call the library.

See the README file in that directory for details on what does each of the programs.

The most complete program in that directory is `sample.cc`, which is a simple version of the `analyzer` program described in section Using `analyzer` Program to Process Corpora with a fixed set of options.

Note that depending on the application, the input text could be obtained from a speech recongnition system, or from a XML parser, or from any source suiting the application goals. Similarly, the obtained analysis, instead of being output, could be used in a translation system, or sent to a dialogue control module, etc.

```
int main (int argc, char **argv) {
  /// set locale to an UTF8 comaptible locale
  util::init_locale(L"default");

  /// path where data files reside
  wstring path=L"/usr/local/share/freeling/es";

  // create analyzers
  tokenizer tk(path+L"tokenizer.dat");
  splitter sp(path+L"splitter.dat");
  splitter::session_id sid=sp.open_session();

  // morphological analysis has a lot of options, and for
simplicity they are
  // packed up in a maco_options object.
  // First, create the maco_options object with default values.
  maco_options opt(L"es");

  // then, provide files for morphological submodules.
  // Note that opt.QuantitiesFile is not set and takes the
default empty value.
  // This will cause quantities module to be deactivated in this
example.
  opt.UserMapFile=L"";
  opt.LocutionsFile=path+L"locucions.dat";
opt.AffixFile=path+L"afixos.dat";
  opt.ProbabilityFile=path+L"probabilitats.dat";
opt.DictionaryFile=path+L"dicc.src";
  opt.NPdataFile=path+L"np.dat";
opt.PunctuationFile=path+L"../common/punct.dat";
  // alternatively, you could set the files in a single call:
```

```
  // opt.set_data_files("", path+"locucions.dat", "",
path+"afixos.dat",
  //                   path+"probabilitats.dat",
opt.DictionaryFile=path+"maco.db",
  //                   path+"np.dat",
path+"../common/punct.dat");

  // create the analyzer with the just build set of maco_options
  maco morfo(opt);
  // then, set required options on/off
  morfo.set_active_options (false, // UserMap
                            true, // NumbersDetection,
                            true, //  PunctuationDetection,
                            true, //  DatesDetection,
                            true, //  DictionarySearch,
                            true, //  AffixAnalysis,
                            false, //  CompoundAnalysis,
                            true, //  RetokContractions,
                            true, //  MultiwordsDetection,
                            true, //  NERecognition,
                            false, //  QuantitiesDetection,
                            true); //  ProbabilityAssignment


  // create a hmm tagger for spanish (with retokenization
ability, and forced
  // to choose only one tag per word)
  hmm_tagger tagger(path+L"tagger.dat", true, FORCE_TAGGER);
  // create chunker
  chart_parser parser(path+L"chunker/grammar-chunk.dat");
  // create dependency parser
  wstring S=parser.get_start_symbol();
  dep_txala dep(path+L"dep_txala/dependences.dat",S);

  // get plain text input lines while not EOF.
  wstring text;
  list<word> lw;
  list<sentence> ls;
  while (getline(wcin,text)) {
```

```
    // tokenize input line into a list of words
    lw=tk.tokenize(text);

    // accumulate list of words in splitter buffer, returning a
list of sentences.
    // The resulting list of sentences may be empty if the
splitter has still not
    // enough evidence to decide that a complete sentence has
been found. The list
    // may contain more than one sentence (since a single input
line may consist
    // of several complete sentences).
    ls=sp.split(sid, lw, false);

    // perform and output morphosyntactic analysis, Pos Tagging
and parsing
    morfo.analyze(ls);
    tagger.analyze(ls);
    parser.analyze(ls);
    dep.analyze(ls);

    // 'ls' contains a list of analyzed sentences.  Do whatever
is needed
    ProcessResults(ls);
    // clear temporary lists;
    lw.clear(); ls.clear();
  }

  // No more lines to read. Make sure the splitter doesn't
retain anything
  sp.split(sid, lw, true, ls);
  sp.close_session(sid);

  // analyze sentence(s) which might be lingering in the buffer,
if any.
  morfo.analyze(ls);
  tagger.analyze(ls);
  parser.analyze(ls);
  dep.analyze(ls);
```

```
  // 'ls' contains a list of analyzed sentences.  Do whatever is
needed
  ProcessResults(ls);

}
```

The processing performed on the obtained results would obviously depend on the goal of the application (translation, indexation, etc.). In order to illustrate the structure of the linguistic data objects, a simple procedure is presented below, in which the processing consists of merely printing the results to stdout in XML format.

```
void ProcessResults(const list<sentence> &ls) {

  list<sentence>::const_iterator is;
  word::const_iterator a;   //iterator over all analysis of a
word
  sentence::const_iterator w;

  // for each sentence in list
  for (is=ls.begin(); is!=ls.end(); is++) {

    wcout<<L"<SENT>"<<endl;
    // for each word in sentence
    for (w=is->begin(); w!=is->end(); w++) {

      // print word form, with PoS and lemma chosen by the
tagger
      wcout<<L"  <WORD form=\""<<w->get_form();
      wcout<<L"\" lemma=\""<<w->get_lemma();
      wcout<<L"\" pos=\""<<w->get_tag();
      wcout<<L"\">"<<endl;

      // for each possible analysis in word, output lemma, tag
and probability
      for (a=w->analysis_begin(); a!=w->analysis_end(); ++a) {
         // print analysis info
         wcout<<L"    <ANALYSIS lemma=\""<<a->get_lemma();
         wcout<<L"\" pos=\""<<a->get_tag();
         wcout<<L"\" prob=\""<<a->get_prob();
         wcout<<L"\"/>"<<endl;
      }

      // close word XML tag after list of analysis
      wcout<<L"  </WORD>"<<endl;
    }

    // close sentence XML tag
    wcout<<L"</SENT>"<<endl;
  }
}
```

The above sample program may be found in `src/main/simple_examples/sample.cc<` in FreeLing tarball. The actual program also outputs tree structures resulting from parsing, which is ommitted here for simplicity.

Once you have compiled and installed FreeLing, you can build this sample program (or any other you may want to write) with the command: `g++ -o sample sample.cc -lfreeling`

Option `-lfreeling` links with libfreeling library, which is the final result of the FreeLing compilation process. Check the README file in the directory to learn more about compiling and using the sample programs.

You may need to add some `-I` and/or `-L` options to the compilation command depending on where the headers and code of required libraries are located. For instance, if you installed some of the libraries in `/usr/local/mylib` instead of the default place `/usr/local`, you'll have to add the options `-I/usr/local/mylib/include -L/usr/local/mylib/lib` to the command above.

Executing `make` in `/src/main/simple_examples` will compile all sample programs in that directory. Make sure that the paths to FreeLing installation directory in `Makefile` are the right ones.

Adding Support for New Languages

It is possible to extend the library with capability to deal with a new language.

In many cases, this may be done simply providing the appropriate linguistic data, with no need to modify any code. Some modules, however, do require entering into the code.

Since the text input language is an configuration option of the system, a new configuration file must be created for the language to be added (e.g. copying and modifying an existing one. The new configuration file must point to the new configuration or data files for each module.

Details on the format for configuration files for each individual module can be found in chapter about Language Processing Modules.

Tokenizer

The first module in the processing chain is the tokenizer. The behaviour of the tokenizer is controlled via the TokenizerFile option in configuration file.

To create a tokenizer for a new language, just create a new tokenization rules file (e.g. copying an existing one and adapting its regexps to particularities of your language), and set it as the value for the TokenizerFile option in your new configuration file.

Morphological analyzer

The morphological analyzer module consists of several sub-modules:

Multiword detection

Just create a multiword file in the appropriate format. The LocutionsFile option in configuration file must be set to the name of a file that contains the multiwords you want to detect in your language.

Nummerical expression detection

If no specialized module is defined to detect nummerical expressions, the default behaviour is to recognize only numbers and codes written in digits (or mixing digits and non-digit characters).

If you want to recognize language dependent expressions (such as numbers expressed in words -e.g. `` `one hundred thirthy-six''), you have to program a `` numbers_mylanguage `class derived from abstract class` numbers_module `. Those classes are finite automata that recognize word sequences. An abstract class` automat` controls the sequence advance, so your derived class has little work to do apart from defining states and transitions for the automaton.

A good idea to start with this issue is having a look at the `numbers_es`, `numbers_en`, `and numbers_ca` classes. State/transition diagrams of those automata can be found in the directory `doc/diagrams`.

Date/time expression detection

If no specialized module is defined to detect date/time expressions, the default behaviour is to recognize only simple date expressions (such as DD/MM/YYYY).

If you want to recognize language dependent expressions (such as complex time expressions -e.g. `` `wednesday, July 12th at half past nine''), you have to program a `` date_mylanguage `class derived from abstract class` dates_module `. Those classes are Augmented Transition Netwrots that recognize word sequences. An abstract class` automat` controls the sequence advance, so your derived class has little work to do apart from defining states and transitions for the automaton.

A good idea to start with this issue is having a look at the `dates_es`, `dates_en`, and `dates_ca` classes. State/transition diagrams of those automata can be found in the directory `doc/diagrams`.

Currency/ratio expression detection

If no specialized module is defined to detect date/time expressions, the default behaviour is to recognize only simple percentage expressions (such as ``23%'').

If you want to recognize language dependent expressions (such as complex ratio expressions -e.g. `three out of four''- or currency expression -e.g.` 2,000 australian dollar''), you have to program a `quantities_mylanguage` class derived from abstract class `quantities_module`. Those classes are Agumented Transition Networks that recognize word sequences. An abstract class `automat` controls the sequence advance, so your derived class has little work to do apart from defining states and transitions for the automaton.

A good idea to start with this issue is having a look at the `quantities_en` and `quantities_es` classes.

In the case your language is a roman language (or at least, has a similar structure for currency expressions) you can easily develop your currency expression detector by copying the `quantities_es` class, and modifying the QuantitiesFile option to provide a file in which lexical items are adapted to your language. For instance: Catalan currency recognizer uses a copy of the `quantities_es` class, but a different QuantitiesFile, since the syntactical structure for currency expression is the same in both languages, but lexical forms are different.

If your language has a very different structure for those expressions, you may require a different format for the QuantitiesFile contents. Since that file will be used only for your language, feel free to readjust its format.

Dictionary search

The lexical forms for each language are sought in a database. You only have to specify in which file it is found with the DictionaryFile option.

The dictionary file can be build with the `src/utilities/dicc-management/build-dict.sh` script included in FreeLing tarball. This program reads all dictionary entry lists given as parameter, and builds a unique file containg a dictionary suitable for FreeLing.

The input files are expected to contain one word form per line, each line with the format:
`form lemma1 tag`
E.g.:

```
abalanzará abalanzar VMIC1S0
abalanzará abalanzar VMIC3S0
bajo bajar VMIP1S0
bajo bajo AQ0MS0
...
```

Affixed forms search

Forms not found in dictionary may be submitted to an affix analysis to devise whether they are derived forms. The valid affixes and their application contexts are defined in the affix rule file referred by AffixFile configuration option.

If your language has ortographic accentuation (such as Spanish, Catalan, and many other roman languages), the suffixation rules may have to deal with accent restoration when rebuilding the original roots. To do this, you have to program a `accents_mylanguage` class

derived from abstract class `accents_module`, which provides the service of restoring (according to the accentuation rules in your languages) accentuation in a root obtained after removing a given suffix.

A good idea to start with this issue is having a look at the `accents_es` class.

Probability assignment

The module in charge of assigning lexical probabilities to each word analysis only requires a data file, referenced by the ProbabilityFile configuration option.

This file may be created using the script `src/utilities/train-tagger/bin/TRAIN.sh` included in FreeLing source package, and a tagged corpus.

HMM PoS Tagger

The HMM PoS tagger only requires an appropriate HMM parameters file, given by the TaggerHMMFile option.

To build a HMM tagger for a new language, you will need corpus (preferably tagged), and you will have to write some probability estimation scripts (e.g. you may use MLE with a simple add-one smoothing).

Nevertheless, the easiest way (if you have a tagged corpus) is using the estimation and smoothing script `src/utilities/train-tagger/bin/TRAIN.sh` provided in FreeLing source package.

See file `src/utilities/train-tagger/README` for details

Relaxation Labelling PoS Tagger

The Relaxation Labelling PoS tagger only requires an appropriate pseudo- constraint grammar file, given by the RelaxTaggerFile option.

To build a Relax tagger for a new language, you will need corpus (preferably tagged), and you will have to write some compatibility estimation scripts. You can also write from scratch a knowledge-based constraint grammar.

Nevertheless, the easiest way (if you have a tagged corpus) is using the estimation and smoothing script `src/utilities/train-tagger/bin/TRAIN.sh` provided in FreeLing source package.

See file `src/utilities/train-tagger/README` for details

Named Entity Recognizer and Classifier

Named Entity recognition and classification modules can be trained for a new language, provided a hand-annotated large enough corpus is available.

See file `src/utilities/train-nrec/README` for details.

Chart Parser

The parser only requires a grammar which is consistent with the tagset used in the morphological and tagging steps. The grammar file must be specified in the GrammarFile option (or passed to the parser constructor).

Rule-based Dependency Parser

The depencency parser only requires a set of rules which is consistent with the PoS tagset and the non-terminal categories generated by the Chart Parser grammar. The grammar file must be specified in the DepRulesFile option (or passed to the parser constructor).

Statistical Dependency Parser and SRL

The statistical depencency parser requires a training corpus with PoS tags matching those produced by the tagger. The corpus needs to be in CoNLL format.

Dependency relations are the output of the module, so they can be any. Of course, later modules (SRL, coreference) will need to be trained or configured to match the labels produced by the dependency parser.

Using `analyzer` Program to Process Corpora

The simplest way to use the FreeLing libraries is via the provided `analyzer` main program, which allows the user to process an input text to obtain several linguistic processings.

Since it is impossible to write a program that fits everyone's needs, `analyzer` offers you almost all functionalities included in FreeLing, but if you want it to output more information, or do so in a specific format, or combine the modules in a different way, the right path to follow is building your own main program or adapting one of the existing, as described in section Using the library from your own application.

The `analyzer` program is usually called with an option `-f config-file` (if ommitted, it will search for a file named `analyzer.cfg` in the current directory). The given `config-file` must be an absolute file name, or a relative path to the current directory.

You can use the default configuration files (located at `/usr/local/share/freeling/config` if you installed from tarball, or at `/usr/share/freeling/config` if you used a `.deb` package), or either a config file that suits your needs. Note that the default configuration files require the environment variable `FREELINGSHARE` to be defined and to point to a directory with valid FreeLing data files (e.g. `/usr/local/share/freeling`).

Environment variables are used for flexibility (e.g. avoid having to modify configuration files if you relocate your data files), but if you don't need them, you can replace all occurrences of `FREELINGSHARE` in your configuration files with a static path.

The `analyzer` program provides also a server mode (use option `-server`) which expects the input from a socket. The program `analyzer_client` can be used to read input files and send requests to the server. The advantatge is that the server remains loaded after analyzing each client's request, thus reducing the start-up overhead if many small files have to be processed. Client and server communicate via sockets. The client-server approach is also a good strategy to call FreeLing from a language or platform for which no API is provided: Just launch a server and use you preferred language to program a client that behaves like `analyzer_client` .

The `analyze` (no final "`r`") script described below handles all these default paths and variables and makes everything easier if you want to use the defaults.

The easy way: Using the `analyze` script

To ease the invocation of the program, a script named `analyze` (no final "r") is provided. This is script is able to locate default configuration files, define library search paths, and handle whether you want the client-server or the straight version.

The sample main program is called with the command:

```
analyze [-f config-file] [options]
```

If `-f config-file` is not specified, a file named `analyzer.cfg` is searched in the current working directory.

If `-f config-file` is specified but not found in the current directory, it will be searched in FreeLing installation directory, which is one of:

- `/usr/local/share/freeling/config` if you installed from source
- `/usr/share/freeling/config` if you used a binary `.deb` package).
- `myfreeling/share/freeling/config` if you used `--prefix=myfreeling` option with `./configure` .

Extra options may be specified in the command line to override any settings in `config-file` . See section Valid Options.

Stand-alone mode

The default mode will launch a stand-alone analyzer, which will load the configuration, read input from stdin, write results to stdout, and exit. E.g.:

```
analyze -f en.cfg <myinput >myoutput
```

When the input file ends, the analyzer will stop and it will have to be reloaded again to process a new file.

Client/server mode

If `--server` and `--port` options are specified, a server will be launched which starts listening for incoming requests. E.g.:

```
analyze -f en.cfg --server --port 50005 &
```

Once the server is launched, clients can request analysis to the server, with:

```
analyzer_client 50005 <myinput >myoutput
analyzer_client localhost:50005 <myinput >myoutput
```

or, from a remote machine:

```
analyzer_client my.server.com:50005 <myinput >myoutput
analyzer_client 192.168.10.11:50005 <myinput >myoutput
```

The server will fork a new process to attend each new client, so you can have many clients being served at the same time.

You can control the maximum amount of clients being attended simutaneously (in order to prevent a flood in your server) with the option `--workers`. You can control the size of the queue of pending clients with option `--queue`. Clients trying to connect when the queue is full will receive a connection error. See section Valid Options for details on these options.

Using a threaded analyzer

If `libboost_thread` is installed, the installation process will build the program `threaded_analyzer`. This program behaves like `analyzer`, and has almost the same options.

The program `threaded_analyzer` launches each processor in a separate thread, so while one sentece is being parsed, the next is being tagged, and the following one is running through the morphological analyzer. In this way, the multi-core capabilities of the host are better exploited and the analyzer runs faster.

Although it is intended mainly as an example for developers wanting to build their own threaded applications, this program can also be used to analyze texts, in the same way than `analyzer`.

Nevertheless, notice that this example program does not include modules that are not token- or sentence-oriented, namely, language identification and coreference resolution.

Usage example

Assuming we have the following input file `mytext.txt`:

```
El gato come pescado. Pero a Don Jaime no le gustan los gatos.
```

we could issue the command:

```
analyze -f myconfig.cfg <mytext.txt >mytext.mrf
```

Assuming that `myconfig.cfg` is the file presented in section Sample Configuration File, the produced output would correspond to a `morfo` output level (i.e. morphological analysis but no PoS tagging). The expected results are:

```
 El el DA0MS0 1
 gato gato NCMS000 1
 come comer VMIP3S0 0.75 comer VMM02S0 0.25
 pescado pescado NCMS000 0.833333 pescar VMP00SM 0.166667
 . . Fp 1

 Pero pero CC 0.99878 pero NCMS000 0.00121951 Pero NP00000
0.00121951
 a a NCFS000 0.0054008 a SPS00 0.994599
 Don_Jaime Don_Jaime NP00000 1
 no no NCMS000 0.00231911 no RN 0.997681
 le él PP3CSD00 1
 gustan gustar VMIP3P0 1
 los el DA0MP0 0.975719 lo NCMP000 0.00019425 él PP3MPA00
0.024087
 gatos gato NCMP000 1
 . . Fp 1
```

If we also wanted PoS tagging, we could have issued the command:

```
analyze -f myconfig.cfg --outlv tagged <mytext.txt >mytext.tag
```

to obtain the tagged output:

```
 El el DA0MS0
 gato gato NCMS000
 come comer VMIP3S0
 pescado pescado NCMS000
 . . Fp

 Pero pero CC
 a a SPS00
 Don_Jaime Don_Jaime NP00000
 no no RN
 le él PP3CSD00
 gustan gustar VMIP3P0
 los el DA0MP0
 gatos gato NCMP000
 . . Fp
```

We can also ask for the senses of the tagged words:

```
analyze -f myconfig.cfg --outlv tagged --sense all <mytext.txt >mytext.sen
```

obtaining the output:

```
El el DA0MS0
 gato gato NCMS000 01630731:07221232:01631653
 come comer VMIP3S0 00794578:00793267
 pescado pescado NCMS000 05810856:02006311
 . . Fp

 Pero pero CC
 a a SPS00
 Don_Jaime Don_Jaime NP00000
 no no RN
 le él PP3CSD00
 gustan gustar VMIP3P0 01244897:01213391:01241953
 los el DA0MP0
 gatos gato NCMP000 01630731:07221232:01631653
 . . Fp
```

Alternatively, if we don't want to repeat the first steps that we had already performed, we could use the output of the morphological analyzer as input to the tagger:

```
analyze -f myconfig.cfg --inplv morfo --outlv tagged <mytext.mrf >mytext.tag
```

See options `InputLevel`, `OutputLevel`, `InputFormat`, and `OutputFormat` in section Valid options for details on which are valid input and output levels and formats.

Configuration File and Command Line Options

Almost all options may be specified either in the configuration file or in the command line, having the later precedence over the former.

Valid options are presented in section Valid options, both in their command-line and configuration file notations. Configuration files follow the usual linux standards. A sample file may be seen in section Sample Configuration File.

FreeLing package includes default configuration files. They can be found at the directory `share/FreeLing/config` under the FreeLing installation directory (`/usr/local` if you installed from source, and `/usr/share/FreeLing` if you used a binary `.deb` package). The `analyze`

script will try to locate the configuration file in that directory if it is not found in the current working directory.

Valid Options

This section presents the options that can be given to the `analyzer` program (and thus, also to the `analyzer_server` program and to the `analyze` script). All options can be written in the configuration file as well as in the command line. The later has always precedence over the former.

Help

Command line	Configuration file
`-h` , `--help` , `--help-cf`	N/A

Prints to stdout a help screen with valid options and exits.
`--help` provides information about command line options.
`--help-cf` provides information about configuration file options.

Version number

Command line	Configuration file
`-v` , `--version`	N/A

Prints the version number of currently installed FreeLing library.

Configuration file

Command line	Configuration file
`-f <filename>`	N/A

Server mode

Command line	Configuration file
`--server`	`ServerMode=(yes/y/on/no/n/off)`

Activate server mode. Requires that option `--port` is also provided.
Default value is `off`.

Server Port Number

Command line	Configuration file
`-p <int>`, `--port <int>`	`ServerPort=<int>`

Specify port where server will be listening for requests. This option must be specified if server mode is active, and it is ignored if server mode is off.

Maximum Number of Server Workers

Command line	Configuration file
`-w <int>`, `--workers <int>`	`ServerMaxWorkers=<int>`

Specify maximum number of active workers that the server will launch. Each worker attends a client, so this is the maximum number of clients that are simultaneously attended. This option is ignored if server mode is off.

Default vaule is 5. Note that a high number of simultaneous workers will result in forking that many processes, which may overload the CPU and memory of your machine resulting in a system collapse.

When the maximum number of workers is reached, new incoming requests are queued until a worker finishes.

Maximum Size of Server Queue

Command line	Configuration file
`-q <int>`, `--queue <int>`	`ServerQueueSize=<int>`

Specify maximum number of pending clients that the server socket can hold. This option is ignored if server mode is off.

Pending clients are requests waiting for a worker to be available. They are queued in the operating system socket queue.

Default value is 32. Note that the operating system has an internal limit for the socket queue size (e.g. modern linux kernels set it to 128). If the given value is higher than the operating system limit, it will be ignored.

When the pending queue is full, new incoming requests get a connection error.

Trace Level

Command line	Configuration file
`-l <int>` , `--tlevel <int>`	`TraceLevel=<int>`

Set the trace level (0 = no trace, higher values = more trace), for debugging purposes.

This will work only if the library was compiled with tracing information, using `./configure -enable-traces`. Note that the code with tracing information is slower than the code compiled without it, even when traces are not active.

Trace Module

Command line	Configuration file
`-m <mask>` , `--tmod <mask>`	`TraceModule=<mask>`

Specify modules to trace. Each module is identified with an hexadecimal flag. All flags may be OR-ed to specificy the set of modules to be traced.

Valid masks are defined in file `src/include/freeling/morfo/traces.h` , and are the following:

Module	Mask
Splitter	`0x00000001`
Tokenizer	`0x00000002`
Morphological analyzer	`0x00000004`
Language Identifier	`0x00000008`
Numbers detection	`0x00000010`
Date/time detection	`0x00000020`
Punctuation	`0x00000040`
Dictionary	`0x00000080`
Affixes	`0x00000100`

Multiwords	`0x00000200`
NE Recognition	`0x00000400`
Probabilities	`0x00000800`
Quantities detection	`0x00001000`
NE Classification	`0x00002000`
Automat (abstract)	`0x00004000`
PoS Tagger	`0x00008000`
Sense annotation	`0x00010000`
Chart parser	`0x00020000`
Chart grammar	`0x00040000`
Dependency parser	`0x00080000`
Coreference resolution	`0x00100000`
Basic utilities	`0x00200000`
WSD	`0x00400000`
Alternatives	`0x00800000`
Database access	`0x01000000`
Feature Extraction	`0x02000000`
Machine Learning modules	`0x04000000`
Phonetic encoding	`0x08000000`
Mention detection	`0x10000000`
Input/Output	`0x20000000`
Semantic graph extraction	`0x40000000`
Summarizer	`0x80000000`

Language of input text

Command line	Configuration file
`--lang <language>`	`Lang=<language>`

Code for language of input text. Though it is not required, the convention is to use two-letter ISO codes (as: Asturian, es: Spanish, ca: Catalan, en: English, cy: Welsh, it: Italian, gl: Galician, pt: Portuguese, ru: Russian, old-es: old Spanish, etc).

Other languages may be added to the library. See chapter Adding Support for New Languages for details.

Locale

Command line	Configuration file
`--locale <locale>`	`Locale=<locale>`

Locale to be used to interpret both input text and data files. Usually, the value will match the locale of the `Lang` option (e.g. `es_ES.utf8` for spanish, `ca_ES.utf8` for Catalan, etc.). The values `default` (stands for `en_US.utf8`) and `system` (stands for currently active system locale) may also be used.

Splitter Buffer Flushing

Command line	Configuration file
`--flush`, `--noflush`	`AlwaysFlush=(yes/y/on/no/n/off)`

When this option is inactive (most usual choice) sentence splitter buffers lines until a sentence marker is found. Then, it outputs a complete sentence.

When this option is active, the splitter never buffers any token, and considers each newline as a sentence end, thus processing each line as an independent sentence.

Input Format

Command line	Configuration file
`--input <string>`	`InputFormat=<string>`

Input format in which to expect text to analyze.

Valid values are:

- `text`: Plain text.
- `freeling`: pseudo-column format produced by freeling with output level `morfo` or `tagged`.
- `conll`: CoNLL-like column format.

Output Format

Command line	Configuration file
`--output <string>`	`OutputFormat=<string>`

Output format to produce with analysis results.

Valid values are:

- `freeling`: Classical freeling format. It may be a pseudo-column for with output levels `morfo` or `tagged`, parenthesized trees for parsing output, or other human-readable output for coreferences or semantic graph output.
- `conll`: CoNLL-like column format.
- `xml`: FreeLing-specific XML format.
- `json`: JSON format
- `naf`: XML format following NAF conventions (see https://github.com/newsreader/NAF)
- `train`: Produce freeling pseudo-column format suitable to train PoS taggers. This option can be used to annotate a corpus, correct the output manually, and use it to retrain the taggers with the script `src/utilities/train-tagger/bin/TRAIN.sh` provided in FreeLing package. See `src/utilities/train-tagger/README` for details about how to use it.

Input Level

Command line	Configuration file
`--inplv <string>`	`InputLevel=<string>`

Analysis level of input data (plain, token, splitted, morfo, tagged, shallow, dep, coref).

- plain: plain text.
- token: tokenized text (one token per line).
- splitted : tokenized and sentence-splitted text (one token per line, sentences separated with one blank line).
- morfo: tokenized, sentence-splitted, and morphologically analyzed text. One token per line, sentences separated with one blank line. Each line has the format: `word (lemma tag prob)`$^{+}$
- tagged: tokenized, sentence-splitted, morphologically analyzed, and PoS-tagged text. One token per line, sentences separated with one blank line. Each line has the format: `word lemma tag`.
- shallow: the previous plus constituency parsing. Only valid with `InputFormat=conll`.

- dep: the previous plus dependency parsing (may include constituents or not. May include also SRL). Only valid with `InputFormat=conll`.
- coref: the previous plus coreference. Only valid with `InputFormat=conll`.

Output Level

Command line	Configuration file
`--outlv <string>`	`OutputLevel=<string>`

Analysis level of output data (ident, token, splitted, morfo, tagged, shallow, dep, coref, semgraph).

- ident: perform language identification instead of analysis.
- token: tokenized text (one token per line).
- splitted : tokenized and sentence-splitted text (one token per line, sentences separated with one blank line).
- morfo: tokenized, sentence-splitted, and morphologically analyzed text. One token per line, sentences separated with one blank line.
- tagged: tokenized, sentence-splitted, morphologically analyzed, and PoS-tagged text. One token per line, sentences separated with one blank line.
- shallow: tokenized, sentence-splitted, morphologically analyzed, PoS-tagged, optionally sense-annotated, and shallow-parsed text, produced by the `chart_parser` module.
- parsed: tokenized, sentence-splitted, morphologically analyzed, PoS-tagged, optionally sense-annotated, and full-parsed text, as output by the first stage (tree completion) of the rule-based dependency parser.
- dep: tokenized, sentence-splitted, morphologically analyzed, PoS-tagged, optionally sense-annotated, and dependency-parsed text, as output by the second stage (transformation to dependencies and function labelling) of the dependency parser. May include also SRL if the statistical parser is used (and SRL is available for the input language).
- coref: the previous plus coreference.
- semgraph: the previous plus semantic graph. Only valid with `OutputFormat=xml|json|freeling`.

Language Identification Configuration File

Command line	Configuration file
`-I <filename>` , `--fidn <filename>`	`N/A`

Configuration file for language identifier.

Tokenizer File

Command line	Configuration file
`--ftok <filename>`	`TokenizerFile=<filename>`

File of tokenization rules.

Splitter File

Command line	Configuration file
`--fsplit <filename>`	`SplitterFile=<filename>`

File of splitter rules.

Affix Analysis

Command line	Configuration file
`--afx`, `--noafx`	`AffixAnalysis=(yes/y/on/no/n/off)`

Whether to perform affix analysis on unknown words. Affix analysis applies a set of affixation rules to the word to check whether it is a derived form of a known word.

Affixation Rules File

Command line	Configuration file
`-S <filename>`, `--fafx <filename>`	`AffixFile=<filename>`

Affix rules file, used by dictionary module.

User Map

Command line	Configuration file
`--usr`, `--nousr`	`UserMap=(yes/y/on/no/n/off)`

Whether to apply or not a file of customized word-tag mappings.

User Map File

Command line	Configuration file
`-M <filename>` , `--fmap <filename>`	`UserMapFile=<filename>`

User Map file to be used.

Multiword Detection

Command line	Configuration file
`--loc` , `--noloc`	`MultiwordsDetection=(yes/y/on/no/n/off)`

Whether to perform multiword detection. This option requires that a multiword file is provided.

Multiword File

Command line	Configuration file
`-L <filename>` , `--floc <filename>`	`LocutionsFile=<filename>`

Multiword definition file.

Number Detection

Command line	Configuration file
`--numb` , `--nonumb`	`NumbersDetection=(yes/y/on/no/n/off)`

Whether to perform nummerical expression detection. Deactivating this feature will affect the behaviour of date/time and ratio/currency detection modules.

Decimal Point

Command line	Configuration file
`--dec <string>`	`DecimalPoint=<string>`

Specify decimal point character for the number detection module (for instance, in English is a dot, but in Spanish is a comma).

Thousand Point

Command line	Configuration file
`--thou <string>`	`ThousandPoint=<string>`

Specify thousand point character for the number detection module (for instance, in English is a comma, but in Spanish is a dot).

Punctuation Detection

Command line	Configuration file
`--punt` , `--nopunt`	`PunctuationDetection=(yes/y/on/no/n/off)`

Whether to assign PoS tag to punctuation signs.

Punctuation Detection File

Command line	Configuration file
`-F <filename>` , `--fpunct <filename>`	`PunctuationFile=<filename>`

Punctuation symbols file.

Date Detection

Command line	Configuration file
`--date` , `--nodate`	`DatesDetection=(yes/y/on/no/n/off)`

Whether to perform date and time expression detection.

Quantities Detection

Command line	Configuration file
`--quant` , `--noquant`	`QuantitiesDetection=(yes/y/on/no/n/off)`

Whether to perform currency amounts, physical magnitudes, and ratio detection.

Quantity Recognition File

Command line	Configuration file
`-Q <filename>` , `--fqty <filename>`	`QuantitiesFile=<filename>`

Quantitiy recognition configuration file.

Dictionary Search

Command line	Configuration file
`--dict` , `--nodict`	`DictionarySearch=(yes/y/on/no/n/off)`

Whether to search word forms in dictionary. Deactivating this feature also deactivates AffixAnalysis option.

Dictionary File

Command line	Configuration file
`-D <filename>` , `--fdict <filename>`	`DictionaryFile=<filename>`

Dictionary database.

Probability Assignment

Command line	Configuration file
`--prob` , `--noprob`	`ProbabilityAssignment=(yes/y/on/no/n/off)`

Whether to compute a lexical probability for each tag of each word. Deactivating this feature will affect the behaviour of the PoS tagger.

Lexical Probabilities File

Command line	Configuration file
`-P <filename>` , `--fprob <filename>`	`ProbabilityFile=<filename>`

Lexical probabilities file. The probabilities in this file are used to compute the most likely tag for a word, as well to estimate the likely tags for unknown words.

Unknown Words Probability Threshold.

Command line	Configuration file
`-e <float>` , `--thres <float>`	`ProbabilityThreshold=<float>`

Threshold that must be reached by the probability of a tag given the suffix of an unknown word in order to be included in the list of possible tags for that word. Default is zero (all tags are included in the list). A non-zero value (e.g. 0.0001, 0.001) is recommended.

Named Entity Recognition

Command line	Configuration file
`--ner` , `--noner`	`NERecognition=(yes/y/on/no/n/off)`

Whether to perform NE recognition.

Named Entity Recognizer File

Command line	Configuration file
`-N <filename>` , `--fnp <filename>`	`NPDataFile=<filename>`

Configuration data file for NE recognizer.

Named Entity Classification

Command line	Configuration file
`--nec` , `--nonec`	`NEClassification=(yes/y/on/no/n/off)`

Whether to perform NE classification.

Named Entity Classifier File

Command line	Configuration file
`--fnec <filename>`	`NECFile=<filename>`

Configuration file for Named Entity Classifier module

Phonetic Encoding

Command line	Configuration file
`--phon` , `--nophon`	`Phonetics=(yes/y/on/no/n/off)`

Whether to add phonetic transcription to each word.

Phonetic Encoder File

Command line	Configuration file
`--fphon <filename>`	`PhoneticsFile=<filename>`

Configuration file for phonetic encoding module

Sense Annotation

Command line	Configuration file
`-s <string>` , `--sense <string>`	`SenseAnnotation=<string>`

Kind of sense annotation to perform

- no, none: Deactivate sense annotation.
- all: annotate with all possible senses in sense dictionary.
- mfs: annotate with most frequent sense.
- ukb: annotate all senses, ranked by UKB algorithm.

Whether to perform sense anotation.

If active, the PoS tag selected by the tagger for each word is enriched with a list of all its possible WN synsets. The sense repository used depends on the options `Sense Annotation Configuration File'' and` UKB Word Sense Disambiguator Configuration File" described below.

Sense Annotation Configuration File

Command line	Configuration file
`-W <filename>` , `--fsense <filename>`	`SenseConfigFile=<filename>`

Word sense annotator configuration file.

UKB Word Sense Disambiguator Configuration File

Command line	Configuration file
`-U <filename>` , `--fukb <filename>`	`UKBConfigFile=<filename>`

UKB configuration file.

Tagger algorithm

Command line	Configuration file
`-t <string>` , `--tag <string>`	`Tagger=<string>`

Algorithm to use for PoS tagging

- hmm: Hidden Markov Model tagger, based on [Bra00].
- relax: Relaxation Labelling tagger, based on [Pad98].

HMM Tagger configuration File

Command line	Configuration file
`-H <filename>` , `--hmm <filename>`	`TaggerHMMFile=<filename>`

Parameters file for HMM tagger.

Relaxation labelling tagger constraints file

Command line	Configuration file
`-R <filename>` , `--rlx <filename>`	`TaggerRelaxFile=<filename>`

File containing the constraints to apply to solve the PoS tagging.

Relaxation labelling tagger iteration limit

Command line	Configuration file
`-i <int>` , `--iter <int>`	`TaggerRelaxMaxIter=<int>`

Maximum numbers of iterations to perform in case relaxation does not converge.

Relaxation labelling tagger scale factor

Command line	Configuration file
`-r <float>` , `--sf <float>`	`TaggerRelaxScaleFactor=<float>`

Scale factor to normalize supports inside RL algorithm. It is comparable to the step lenght in a hill-climbing algorithm: The larger scale factor, the smaller step.

Relaxation labelling tagger epsilon value

Command line	Configuration file
`--eps <float>`	`TaggerRelaxEpsilon=<float>`

Real value used to determine when a relaxation labelling iteration has produced no significant changes. The algorithm stops when no weight has changed above the specified epsilon.

Retokenize contractions in dictionary

Command line	Configuration file
`--rtkcon` , `--nortkcon`	`RetokContractions=(yes/y/on/no/n/off)`

Specifies whether the dictionary must retokenize contractions when found, or leave the decision to the `TaggerRetokenize` option.

Note that if this option is active, contractions will be retokenized even if the `TaggerRetokenize` option is not active. If this option is not active, contractions will be retokenized depending on the value of the `TaggerRetokenize` option.

Retokenize after tagging

Command line	Configuration file
`--rtk` , `--nortk`	`TaggerRetokenize=(yes/y/on/no/n/off)`

Determine whether the tagger must perform retokenization after the appropriate analysis has been selected for each word. This is closely related to affix analysis and PoS taggers.

Force the selection of one unique tag

Command line	Configuration file
`--force <string>`	`TaggerForceSelect=(none/tagger/retok)`

Determine whether the tagger must be forced to (probably randomly) make a unique choice and when.

- `none`: Do not force the tagger, allow ambiguous output.
- `tagger`: Force the tagger to choose before retokenization (i.e. if retokenization introduces any ambiguity, it will be present in the final output).
- `retok`: Force the tagger to choose after retokenization (no remaining ambiguity)

Chart Parser Grammar File

Command line	Configuration file
`-G <filename>` , `--grammar <filename>`	`GrammarFile=<filename>`

This file contains a CFG grammar for the chart parser, and some directives to control which chart edges are selected to build the final tree.

Dependency Parser Rule File

Command line	Configuration file
`-T <filename>` , `--txala <filename>`	`DepTxalaFile==<filename>`

Rules to be used to perform rule-based dependency analysis.

Statistical Dependency Parser File

Command line	Configuration file
`-E <filename>` , `--treeler <filename>`	`DepTreelerFile==<filename>`

Configuration file for statistical dependency parser and SRL module

---

Dependency Parser Selection

Command line	Configuration file
`-d <string>` , `--dep <string>`	`DependencyParser==<string>`

Which dependency parser to use. Valid values are:

- `txala` (rule-based)
- `treeler` (statistical, may have SRL also).

Coreference Resolution File

Command line	Configuration file
`-C <filename>` , `--fcorf <filename>`	`CorefFile=<filename>`

Configuration file for coreference resolution module.

Sample Configuration File

A sample configuration file follows. You can start using freeling with the default configuration files which are installed at `/usr/local/share/freeling/config` (note than prefix `/usr/local` may differ if you specified an alternative location when installing FreeLing. If you installed from a binary `.deb` package), it will be at `/usr/share/freeling/config` .

You can use those files as a starting point to customize one configuration file to suit your needs.

Note that file paths in the sample configuration file contain `$FREELINGSHARE` , which is supposed to be an environment variable. If this variable is not defined, the analyzer will abort, complaining about not finding the files.

If you use the `analyze` script, it will define the variable for you as `/usr/local/share/Freeling` (or the right installation path), unless you define it to point somewhere else.

You can also adjust your configuration files to use normal paths for the files (either relative or absolute) instead of using variables.

```
##
#### default configuration file for Spanish analyzer
##

#### General options
Lang=es
Locale=default

### Tagset description file, used by different modules
TagsetFile=$FREELINGSHARE/es/tagset.dat

## Traces (deactivated)
TraceLevel=0
TraceModule=0x0000

## Options to control the applied modules. The input may be
partially
## processed, or not a full analysis may me wanted. The specific
## formats are a choice of the main program using the library,
as well
## as the responsability of calling only the required modules.
InputLevel=text
OutputLevel=morfo

# Do not consider each newline as a sentence end
AlwaysFlush=no

#### Tokenizer options
TokenizerFile=$FREELINGSHARE/es/tokenizer.dat

#### Splitter options
SplitterFile=$FREELINGSHARE/es/splitter.dat

#### Morfo options
AffixAnalysis=yes
CompoundAnalysis=yes
MultiwordsDetection=yes
```

```
NumbersDetection=yes
PunctuationDetection=yes
DatesDetection=yes
QuantitiesDetection=yes
DictionarySearch=yes
ProbabilityAssignment=yes
DecimalPoint=,
ThousandPoint=.
LocutionsFile=$FREELINGSHARE/es/locucions.dat
QuantitiesFile=$FREELINGSHARE/es/quantities.dat
AffixFile=$FREELINGSHARE/es/afixos.dat
CompoundFile=$FREELINGSHARE/es/compounds.dat
ProbabilityFile=$FREELINGSHARE/es/probabilitats.dat
DictionaryFile=$FREELINGSHARE/es/dicc.src
PunctuationFile=$FREELINGSHARE/common/punct.dat
ProbabilityThreshold=0.001

# NER options
NERecognition=yes
NPDataFile=$FREELINGSHARE/es/np.dat
## comment line above and uncomment one of those below, if you
want
## a better NE recognizer (higer accuracy, lower speed)
#NPDataFile=$FREELINGSHARE/es/nerc/ner/ner-ab-poor1.dat
#NPDataFile=$FREELINGSHARE/es/nerc/ner/ner-ab-rich.dat
# "rich" model is trained with rich gazetteer. Offers higher
accuracy but
# requires adapting gazetteer files to have high coverage on
target corpus.
# "poor1" model is trained with poor gazetteer. Accuracy is
splightly lower
# but suffers small accuracy loss the gazetteer has low coverage
in target corpus.
# If in doubt, use "poor1" model.

## Phonetic encoding of words.
Phonetics=no
PhoneticsFile=$FREELINGSHARE/es/phonetics.dat

## NEC options. See README in common/nec
```

```
NEClassification=no
NECFile=$FREELINGSHARE/es/nerc/nec/nec-ab-poor1.dat
#NECFile=$FREELINGSHARE/es/nerc/nec/nec-ab-rich.dat

## Sense annotation options (none,all,mfs,ukb)
SenseAnnotation=none
SenseConfigFile=$FREELINGSHARE/es/senses.dat
UKBConfigFile=$FREELINGSHARE/es/ukb.dat

#### Tagger options
Tagger=hmm
TaggerHMMFile=$FREELINGSHARE/es/tagger.dat
TaggerRelaxFile=$FREELINGSHARE/es/constr_gram-B.dat
TaggerRelaxMaxIter=500
TaggerRelaxScaleFactor=670.0
TaggerRelaxEpsilon=0.001
TaggerRetokenize=yes
TaggerForceSelect=tagger

#### Parser options
GrammarFile=$FREELINGSHARE/es/chunker/grammar-chunk.dat

#### Dependence Parser options
DependencyParser=txala
DepTxalaFile=$FREELINGSHARE/es/dep_txala/dependences.dat
DepTreelerFile=$FREELINGSHARE/es/dep_treeler/dependences.dat

#### Coreference Solver options
CorefFile=$FREELINGSHARE/es/coref/relaxcor/relaxcor.dat
SemGraphExtractorFile=$FREELINGSHARE/es/semgraph/semgraph-
SRL.dat
```

FreeLing Tagset Description

FreeLing morphological analyzers and taggser encode morphological information in PoS tags which are based on the proposals by EAGLES.

EAGLES intends to be able to encode all existing morphological features for most European languages.

EAGLES PoS tags consist of variable-length labels where each character corresponds to a morphological feature. First character in the tag is always the category (PoS). The category determines the length of the tag and the interpretation of each character in the label.

For instance, for category `noun` we could have the definition:

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*f*; **M**:*m*; **C**:*c*
3	num	**S**:*s*; **P**:*p*; **N**:*n*

That would allow PoS tags such as `NCMS` (standing for *noun/common/masculine/singular*)

Features that are not applicable or underspecified for a particular word are set to `0` (zero). For instance the tag `NC00` stands for *noun/common/underspecified-gender/underspecified-number*.

Note that the interpretation of a character at a certain position of a tag depends on the PoS (indicated by the first character) and on the target language.

For instance, in a language where nouns can have additional features (e.g. case) the tag definition would include one additional position for case feature. E.g.:

Position	Attribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **F**:*accusative*;
3	gen	**F**:*f*; **M**:*m*; **C**:*c*
4	num	**S**:*s*; **P**:*p*; **N**:*n*

The order of the features is decided by the tagset designer, although EAGLES guidelines recommend that most general and most informative features are placed at the beginning of the label. In this way, labels can be shortened to a prefix that will always keep the essence of the PoS tag (being the extreme case the reduction to one single character that would be the category).

Next sections contain valid Part-of-Speech tags and attributes with their values for each language supported in FreeLing:

- (as) Asturian
- (ca) Catalan
- (cy) Welsh
- (de) German
- (en) English
- (es) Spanish
- (fr) French
- (gl) Galician
- (hr) Croatian
- (it) Italian
- (nb) Norwegian
- (pt) Portuguese
- (ru) Russian
- (sl) Slovene

Tagset for Asturian (as)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*
2	degree	**S**:*superlative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessorpers	**1**:*1*; **2**:*2*; **3**:*3*
6	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*; **A**:*adversative*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*possessive*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**V**:*evaluative*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*personal*; **X**:*possessive*; **R**:*relative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	possessornum	**S**:*singular*; **P**:*plural*
7	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*pastparticiple*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **C**:*conditional*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*ratio*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Catalan (ca)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*; **P**:*possessive*
2	degree	**S**:*superlative*; **V**:*evaluative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessorpers	**1**:*1*; **2**:*2*; **3**:*3*
6	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **I**:*indefinite*; **P**:*possessive*; **R**:*relative*; **T**:*interrogative*; **E**:*exclamative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**V**:*evaluative*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **P**:*personal*; **R**:*relative*; **T**:*interrogative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*participle*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **C**:*conditional*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*percentage*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Welsh (cy)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*
2	degree	**C**:*comparative*; **S**:*superlative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*possessive*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*

Part of Speech: `particle`

Position	Atribute	Values
0	category	**G**:*particle*
1	type	**N**:*negative*; **A**:*article*; **I**:*indefinite*; **V**:*verbal*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**A**:*augmentative*; **D**:*diminutive*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*personal*; **X**:*possessive*; **R**:*relative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	possessornum	**S**:*singular*; **P**:*plural*
7	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*pastparticiple*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **C**:*conditional*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*ratio*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for German (de)

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **G**:*genitive*
3	gen	**F**:*feminine*; **M**:*masculine*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*
5	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**Q**:*qualificative*; **P**:*predicative*
2	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **G**:*genitive*
3	gen	**F**:*feminine*; **M**:*masculine*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*
5	degree	**S**:*superlative*; **C**:*comparative*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*definite*; **I**:*indefinite*; **X**:*possessive*; **D**:*demostrative*; **R**:*relative*; **T**:*interrogative*
2	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **G**:*genitive*
3	gen	**F**:*feminine*; **M**:*masculine*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**V**:*full*; **A**:*auxiliary*; **M**:*modal*
2	mood	**N**:*infinitive*; **P**:*participle*; **Z**:*zuinf*; **M**:*imperative*; **S**:*subjunctive*; **I**:*indicative*
3	tense	**P**:*present*; **S**:*past*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **R**:*relative*; **T**:*interrogative*; **P**:*personal*; **F**:*reflexive*; **X**:*possessive*; **I**:*indefinite*
2	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **G**:*genitive*
3	gen	**F**:*feminine*; **M**:*masculine*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*
5	person	**1**:*1*; **2**:*2*; **3**:*3*
6	politeness	**P**:*polite*; **F**:*familiar*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**G**:*general*; **N**:*negative*; **V**:*particle*; **P**:*pronominal*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*; **S**:*postposition*; **C**:*circumposition*; **Z**:*particle*
2	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **G**:*genitive*
3	gen	**F**:*feminine*; **M**:*masculine*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**S**:*subordinating*; **C**:*coordinating*; **M**:*comparative*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*percentage*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for English (en)

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*percentage*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `adjective`

Tag	Attributes
JJ	**pos**:*adjective*
JJR	**pos**:*adjective*; **degree**:*comparative*
JJS	**pos**:*adjective*; **degree**:*superlative*

Part of Speech: `adposition`

Tag	Attributes
POS	**pos**:*adposition*; **type**:*possessive*

Part of Speech: `adverb`

Tag	Attributes
RB	**pos**:*adverb*; **type**:*general*
RBR	**pos**:*adverb*; **type**:*general*; **degree**:*comparative*
RBS	**pos**:*adverb*; **type**:*general*; **degree**:*superlative*
WRB	**pos**:*adverb*; **type**:*interrogative*

Part of Speech: `conjunction`

Tag	Attributes
CC	**pos**:*conjunction*; **type**:*coordinating*

Part of Speech: `determiner`

Tag	Attributes
DT	**pos**:*determiner*
WDT	**pos**:*determiner*; **type**:*interrogative*
PDT	**pos**:*determiner*; **type**:*predeterminer*

Part of Speech: `interjection`

Tag	Attributes
UH	**pos**:*interjection*

Part of Speech: `noun`

Tag	Attributes
NNS	**pos**:*noun*; **type**:*common*; **num**:*plural*
NN	**pos**:*noun*; **type**:*common*; **num**:*singular*
NNP	**pos**:*noun*; **type**:*proper*
NP00000	**pos**:*noun*; **type**:*proper*
NP	**pos**:*noun*; **type**:*proper*
NP00G00	**pos**:*noun*; **type**:*proper*; **neclass**:*location*
NP00O00	**pos**:*noun*; **type**:*proper*; **neclass**:*organization*
NP00V00	**pos**:*noun*; **type**:*proper*; **neclass**:*other*
NP00SP0	**pos**:*noun*; **type**:*proper*; **neclass**:*person*
NNPS	**pos**:*noun*; **type**:*proper*; **num**:*plural*

Part of Speech: `particle`

Tag	Attributes
RP	**pos**:*particle*
TO	**pos**:*particle*; **type**:*to*

Part of Speech: `preposition`

Tag	Attributes
IN	**pos**:*preposition*

Part of Speech: `pronoun`

Tag	Attributes
EX	**pos**:*pronoun*
WP	**pos**:*pronoun*; **type**:*interrogative*
PRP	**pos**:*pronoun*; **type**:*personal*
PRP$	**pos**:*pronoun*; **type**:*possessive*
WP$	**pos**:*pronoun*; **type**:*possessive*

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Part of Speech: `verb`

Tag	Attributes
MD	**pos**:*verb*; **type**:*modal*
VBG	**pos**:*verb*; **vform**:*gerund*
VB	**pos**:*verb*; **vform**:*infinitive*
VBN	**pos**:*verb*; **vform**:*participle*
VBD	**pos**:*verb*; **vform**:*past*
VBP	**pos**:*verb*; **vform**:*personal*
VBZ	**pos**:*verb*; **vform**:*personal*; **person**:*3*

Tagset for Spanish (es)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*; **P**:*possessive*
2	degree	**S**:*superlative*; **V**:*evaluative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessorpers	**1**:*1*; **2**:*2*; **3**:*3*
6	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **I**:*indefinite*; **P**:*possessive*; **T**:*interrogative*; **E**:*exclamative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**V**:*evaluative*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **P**:*personal*; **R**:*relative*; **T**:*interrogative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*participle*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **C**:*conditional*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*percentage*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for French (fr)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*; **P**:*possessive*
2	degree	**S**:*superlative*; **V**:*evaluative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessorpers	**1**:*1*; **2**:*2*; **3**:*3*
6	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*possessive*; **R**:*relative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**A**:*augmentative*; **D**:*diminutive*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*personal*; **X**:*possessive*; **R**:*relative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*participle*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **C**:*conditional*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*percentage*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolonk*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Galician (gl)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*
2	degree	**S**:*superlative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*possessive*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**V**:*evaluative*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*personal*; **X**:*possessive*; **R**:*relative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	possessornum	**S**:*singular*; **P**:*plural*
7	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*
2	contracted	**C**:*yes*
3	gen	**M**:*masculine*
4	num	**S**:*singular*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*pastparticiple*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **M**:*plusquamperfect*; **C**:*conditional*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*ratio*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Croatian (hr)

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
3	num	**S**:*singular*; **P**:*plural*
4	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **V**:*vocative*; **L**:*locative*; **I**:*instrumental*
5	animate	**N**:*no*; **Y**:*yes*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **O**:*modal*; **C**:*copula*
2	vform	**I**:*indicative*; **M**:*imperative*; **C**:*conditional*; **N**:*infinitive*; **P**:*participle*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **L**:*pluperfect*; **A**:*aorist*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
7	voice	**A**:*active*; **P**:*passive*
8	negative	**N**:*no*; **Y**:*yes*

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**F**:*qualificative*; **S**:*possessive*; **O**:*ordinal*
2	degree	**P**:*positive*; **C**:*comparative*; **S**:*superlative*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*
5	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **V**:*vocative*; **L**:*locative*; **I**:*instrumental*
6	definite	**N**:*no*; **Y**:*yes*
7	animate	**N**:*no*; **Y**:*yes*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**P**:*personal*; **D**:*demonstrative*; **I**:*indefinite*; **S**:*possessive*; **R**:*relative*; **X**:*reflexive*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*
5	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **V**:*vocative*; **L**:*locative*; **I**:*instrumental*
6	definite	**N**:*no*; **Y**:*yes*
7	possessornum	**S**:*singular*; **P**:*plural*
8	possessorgen	**M**:*m*; **F**:*f*; **N**:*n*
9	clitic	**Y**:*yes*; **N**:*no*
10	referent	**P**:*personal*; **S**:*possessive*
11	syntactic	**N**:*nominal*; **A**:*adjectival*
12	animate	**N**:*no*; **Y**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**G**:*general*
2	degree	**P**:*positive*; **C**:*comparative*; **S**:*superlative*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*
2	formation	**S**:*simple*; **C**:*compound*
3	case	**G**:*genitive*; **D**:*dative*; **A**:*accusative*; **L**:*locative*; **I**:*instrumental*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*
2	formation	**S**:*simple*; **C**:*compound*

Part of Speech: `numeral`

Position	Atribute	Values
0	category	**M**:*numeral*
1	type	**C**:*cardinal*; **O**:*ordinal*; **M**:*multiple*; **S**:*special*
2	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
3	num	**S**:*singular*; **P**:*plural*
4	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **V**:*vocative*; **L**:*locative*; **I**:*instrumental*
5	form	**D**:*digit*; **R**:*roman*; **L**:*letter*
6	animate	**N**:*no*; **Y**:*yes*

Part of Speech: `particle`

Position	Atribute	Values
0	category	**Q**:*particle*
1	type	**Z**:*negative*; **Q**:*interrogative*; **O**:*modal*; **R**:*affirmative*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*
1	formation	**S**:*simple*; **C**:*compound*

Part of Speech: `abbreviation`

Position	Atribute	Values
0	category	**Y**:*abbreviation*
1	syntactic	**N**:*nominal*; **A**:*adjectival*
2	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
3	num	**S**:*singular*; **D**:*dual*; **P**:*plural*
4	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **L**:*locative*; **I**:*instrumental*

Part of Speech: `residual`

Position	Atribute	Values
0	category	**X**:*residual*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*percentage*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Italian (it)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*; **P**:*possessive*
2	degree	**S**:*superlative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessorpers	**1**:*1*; **2**:*2*; **3**:*3*
6	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*possessive*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**A**:*augmentative*; **D**:*diminutive*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*personal*; **R**:*relative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*
2	contracted	**C**:*yes*
3	gen	**M**:*masculine*
4	num	**S**:*singular*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*pastparticiple*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **C**:*conditional*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*ratio*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Norwegian (nb)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*
2	degree	**S**:*superlative*; **A**:*comparative*; **P**:*positive*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*
5	function	**P**:*participle*; **R**:*preparticiple*
6	case	**G**:*genitive*
7	definite	**D**:*yes*; **U**:*no*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*; **A**:*adverbial*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**D**:*demonstrative*; **P**:*possessive*; **T**:*interrogative*; **M**:*amplifier*; **Q**:*quantifier*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*
5	definite	**D**:*yes*; **U**:*no*
6	other	**P**:*polite*; **R**:*reciprocal*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
3	num	**S**:*singular*; **P**:*plural*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	case	**N**:*nominative*; **G**:*genitive*
7	definite	**D**:*yes*; **U**:*no*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **Q**:*quantifier*; **T**:*interrogative*; **P**:*personal*; **X**:*possessive*; **R**:*relative*; **C**:*reciprocal*; **F**:*reflexive*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*
5	case	**N**:*nominative*; **A**:*accusative*
6	polite	**P**:*yes*
7	human	**H**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*
2	contracted	*Not used*
3	gen	**M**:*masculine*; **F**:*feminine*
4	num	**S**:*singular*; **P**:*plural*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*; **V**:*sverb*; **P**:*passive*
2	mood	**I**:*indicative*; **M**:*imperative*; **P**:*participle*; **N**:*infinitive*
3	tense	**P**:*present*; **S**:*past*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*percentage*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `particle`

Tag	Attributes
TO	**pos**:*particle*; **type**:*to*

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Portuguese (pt)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**O**:*ordinal*; **Q**:*qualificative*; **P**:*possessive*
2	degree	**S**:*superlative*; **V**:*evaluative*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessorpers	**1**:*1*; **2**:*2*; **3**:*3*
6	possessornum	**S**:*singular*; **P**:*plural*; **N**:*invariable*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `determiner`

Position	Atribute	Values
0	category	**D**:*determiner*
1	type	**A**:*article*; **D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*possessive*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	possessornum	**S**:*singular*; **P**:*plural*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
3	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
4	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*
5	nesubclass	*Not used*
6	degree	**A**:*augmentative*; **D**:*diminutive*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**D**:*demonstrative*; **E**:*exclamative*; **I**:*indefinite*; **T**:*interrogative*; **N**:*numeral*; **P**:*personal*; **R**:*relative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*
4	num	**S**:*singular*; **P**:*plural*; **N**:*invariable*
5	case	**N**:*nominative*; **A**:*accusative*; **D**:*dative*; **O**:*oblique*
6	polite	**P**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**N**:*negative*; **G**:*general*

Part of Speech: `adposition`

Position	Atribute	Values
0	category	**S**:*adposition*
1	type	**P**:*preposition*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*; **S**:*semiauxiliary*
2	mood	**I**:*indicative*; **S**:*subjunctive*; **M**:*imperative*; **P**:*pastparticiple*; **G**:*gerund*; **N**:*infinitive*
3	tense	**P**:*present*; **I**:*imperfect*; **F**:*future*; **S**:*past*; **C**:*conditional*; **M**:*plusquamperfect*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **P**:*plural*
6	gen	**F**:*feminine*; **M**:*masculine*; **C**:*common*; **N**:*neuter*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*ratio*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Russian (ru)

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **F**:*accusative*; **C**:*instrumental*; **O**:*prepositional*; **P**:*partitive*; **L**:*locative*; **V**:*vocative*
2	num	**S**:*singular*; **P**:*plural*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*; **C**:*common*
4	animate	**A**:*yes*; **I**:*no*
5	form	**F**:*full*; **S**:*short*
6	degree	**E**:*superlative*; **C**:*comparative*; **P**:*positive*
7	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*
8	obscene	**H**:*yes*

Part of Speech: `preposition`

Position	Atribute	Values
0	category	**B**:*preposition*
1	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**D**:*adverb*
1	degree	**E**:*superlative*; **C**:*comparative*; **P**:*positive*
2	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*
3	obscene	**H**:*yes*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**E**:*pronoun*
1	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **F**:*accusative*; **C**:*instrumental*; **O**:*prepositional*; **P**:*partitive*; **L**:*locative*; **V**:*vocative*
2	num	**S**:*singular*; **P**:*plural*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*; **C**:*common*
4	animate	**A**:*yes*; **I**:*no*
5	person	**1**:*1*; **2**:*2*; **3**:*3*
6	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**J**:*interjection*
1	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*
2	obscene	**H**:*yes*

Part of Speech: `compound`

Position	Atribute	Values
0	category	**M**:*compound*
1	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **F**:*accusative*; **C**:*instrumental*; **O**:*prepositional*; **P**:*partitive*; **L**:*locative*; **V**:*vocative*
3	num	**S**:*singular*; **P**:*plural*
4	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*; **C**:*common*
5	animate	**A**:*yes*; **I**:*no*
6	info	**G**:*geographical*; **N**:*name*; **S**:*patronymic*; **F**:*surname*
7	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*
8	obscene	**H**:*yes*
9	neclass	**S**:*person*; **G**:*location*; **O**:*organization*; **V**:*other*

Part of Speech: `pronominal-adv`

Position	Atribute	Values
0	category	**P**:*pronominal-adv*
1	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*

Part of Speech: `participle`

Position	Atribute	Values
0	category	**Q**:*participle*
1	mood	**G**:*gerund*; **I**:*infinitive*; **D**:*indicative*; **M**:*imperative*
2	num	**S**:*singular*; **P**:*plural*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*; **C**:*common*
4	tense	**P**:*present*; **F**:*future*; **S**:*past*
5	person	**1**:*1*; **2**:*2*; **3**:*3*
6	aspect	**F**:*perfective*; **N**:*imperfective*
7	voice	**A**:*active*; **P**:*passive*
8	transitive	**M**:*yes*; **A**:*no*
9	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*
10	obscene	**H**:*yes*

Part of Speech: `pronominal-adj`

Position	Atribute	Values
0	category	**R**:*pronominal-adj*
1	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **F**:*accusative*; **C**:*instrumental*; **O**:*prepositional*; **P**:*partitive*; **L**:*locative*; **V**:*vocative*
2	num	**S**:*singular*; **P**:*plural*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*; **C**:*common*
4	animate	**A**:*yes*; **I**:*no*
5	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*

Part of Speech: `particle`

Position	Atribute	Values
0	category	**T**:*particle*
1	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	mood	**G**:*gerund*; **I**:*infinitive*; **D**:*indicative*; **M**:*imperative*
2	num	**S**:*singular*; **P**:*plural*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*; **A**:*ambiguous*
4	tense	**P**:*present*; **F**:*future*; **S**:*past*
5	person	**1**:*1*; **2**:*2*; **3**:*3*
6	aspect	**F**:*perfective*; **N**:*imperfective*
7	voice	**A**:*active*; **S**:*passive*
8	transitive	**M**:*yes*; **A**:*no*
9	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*
10	obscene	**H**:*yes*

Part of Speech: `ordinal`

Position	Atribute	Values
0	category	**Y**:*ordinal*
1	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **F**:*accusative*; **C**:*instrumental*; **O**:*prepositional*; **P**:*partitive*; **L**:*locative*; **V**:*vocative*
2	num	**S**:*singular*; **P**:*plural*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*; **C**:*common*
4	animate	**A**:*yes*; **I**:*no*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **F**:*accusative*; **C**:*instrumental*; **O**:*prepositional*; **P**:*partitive*; **L**:*locative*; **V**:*vocative*
2	num	**S**:*singular*; **P**:*plural*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*; **C**:*common*
4	animate	**A**:*yes*; **I**:*no*
5	other	**P**:*transition*; **D**:*difficult*; **V**:*corrupted*; **R**:*predicative*; **I**:*spoken*; **A**:*uncommon*; **B**:*abbreviation*; **E**:*outdated*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

Tagset for Slovene (sl)

Part of Speech: `noun`

Position	Atribute	Values
0	category	**N**:*noun*
1	type	**C**:*common*; **P**:*proper*
2	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
3	num	**S**:*singular*; **D**:*dual*; **P**:*plural*
4	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **L**:*locative*; **I**:*instrumental*
5	animate	**N**:*no*; **Y**:*yes*

Part of Speech: `verb`

Position	Atribute	Values
0	category	**V**:*verb*
1	type	**M**:*main*; **A**:*auxiliary*
2	aspect	**E**:*perfective*; **P**:*progressive*; **B**:*biaspectual*
3	vform	**N**:*infinitive*; **U**:*supine*; **P**:*participle*; **R**:*present*; **F**:*future*; **C**:*conditional*; **M**:*imperative*
4	person	**1**:*1*; **2**:*2*; **3**:*3*
5	num	**S**:*singular*; **D**:*dual*; **P**:*plural*
6	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
7	negative	**N**:*no*; **Y**:*yes*

Part of Speech: `adjective`

Position	Atribute	Values
0	category	**A**:*adjective*
1	type	**G**:*general*; **S**:*possessive*; **P**:*participle*
2	degree	**P**:*positive*; **C**:*comparative*; **S**:*superlative*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
4	num	**S**:*singular*; **D**:*dual*; **P**:*plural*
5	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **L**:*locative*; **I**:*instrumental*
6	definite	**N**:*no*; **Y**:*yes*

Part of Speech: `adverb`

Position	Atribute	Values
0	category	**R**:*adverb*
1	type	**G**:*general*; **R**:*participle*
2	degree	**P**:*positive*; **C**:*comparative*; **S**:*superlative*

Part of Speech: `pronoun`

Position	Atribute	Values
0	category	**P**:*pronoun*
1	type	**P**:*personal*; **S**:*possessive*; **D**:*demonstrative*; **R**:*relative*; **X**:*reflexive*; **C**:*General*; **Q**:*interrogative*; **I**:*indefinite*; **Z**:*negative*
2	person	**1**:*1*; **2**:*2*; **3**:*3*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
4	num	**S**:*singular*; **D**:*dual*; **P**:*plural*
5	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **L**:*locative*; **I**:*instrumental*
6	possessornum	**S**:*singular*; **D**:*dual*; **P**:*plural*
7	possessorgen	**M**:*m*; **F**:*f*; **N**:*n*
8	clitic	**Y**:*yes*; **B**:*bound*

Part of Speech: `number`

Position	Atribute	Values
0	category	**M**:*number*
1	form	**D**:*digit*; **R**:*roman*; **L**:*letter*
2	type	**C**:*cardinal*; **O**:*ordinal*; **P**:*pronomial*; **S**:*special*
3	gen	**M**:*masculine*; **F**:*feminine*; **N**:*neuter*
4	num	**S**:*singular*; **D**:*dual*; **P**:*plural*
5	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **L**:*locative*; **I**:*instrumental*
6	definite	**N**:*no*; **Y**:*yes*

Part of Speech: `preposition`

Position	Atribute	Values
0	category	**S**:*preposition*
1	case	**N**:*nominative*; **G**:*genitive*; **D**:*dative*; **A**:*accusative*; **L**:*locative*; **I**:*instrumental*

Part of Speech: `conjunction`

Position	Atribute	Values
0	category	**C**:*conjunction*
1	type	**C**:*coordinating*; **S**:*subordinating*

Part of Speech: `particle`

Position	Atribute	Values
0	category	**Q**:*particle*

Part of Speech: `interjection`

Position	Atribute	Values
0	category	**I**:*interjection*

Part of Speech: `abbreviation`

Position	Atribute	Values
0	category	**Y**:*abbreviation*

Part of Speech: `residual`

Position	Atribute	Values
0	category	**X**:*residual*
1	type	**F**:*foreign*; **T**:*typo*; **P**:*program*

Part of Speech: `number`

Position	Atribute	Values
0	category	**Z**:*number*
1	type	**d**:*partitive*; **m**:*currency*; **p**:*percentage*; **u**:*unit*

Part of Speech: `date`

Position	Atribute	Values
0	category	**W**:*date*

Non-positional tags

Part of Speech: `punctuation`

Tag	Attributes
Fd	**pos**:*punctuation*; **type**:*colon*
Fc	**pos**:*punctuation*; **type**:*comma*
Flt	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*close*
Fla	**pos**:*punctuation*; **type**:*curlybracket*; **punctenclose**:*open*
Fs	**pos**:*punctuation*; **type**:*etc*
Fat	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*close*
Faa	**pos**:*punctuation*; **type**:*exclamationmark*; **punctenclose**:*open*
Fg	**pos**:*punctuation*; **type**:*hyphen*
Fz	**pos**:*punctuation*; **type**:*other*
Fpt	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*close*
Fpa	**pos**:*punctuation*; **type**:*parenthesis*; **punctenclose**:*open*
Ft	**pos**:*punctuation*; **type**:*percentage*
Fp	**pos**:*punctuation*; **type**:*period*
Fit	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*close*
Fia	**pos**:*punctuation*; **type**:*questionmark*; **punctenclose**:*open*
Fe	**pos**:*punctuation*; **type**:*quotation*
Frc	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*close*
Fra	**pos**:*punctuation*; **type**:*quotation*; **punctenclose**:*open*
Fx	**pos**:*punctuation*; **type**:*semicolon*
Fh	**pos**:*punctuation*; **type**:*slash*
Fct	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*close*
Fca	**pos**:*punctuation*; **type**:*squarebracket*; **punctenclose**:*open*

References

- **[ACM05]** Jordi Atserias, Elisabet Comelles, and Aingeru Mayor.
  Txala: un analizador libre de dependencias para el castellano.
  Procesamiento del Lenguaje Natural, (35):455-456, September 2005.

- **[AS09]** Eneko Agirre and Aitor Soroa.
  Personalizing pagerank for word sense disambiguation.
  In *Proceedings of the 12th conference of the European chapter of the Association for Computational Linguistics (EACL-2009)*, Athens, Greece, 2009.

- **[Bra00]** Thorsten Brants.
  Tnt: A statistical part-of-speech tagger.
  In *Proceedings of the 6th Conference on Applied Natural Language Processing, ANLP*, 2000.

- **[Car07]** Xavier Carreras.
  Experiments with a Higher-Order Projective Dependency Parser.
  In Proceedings of the EMNLP-CoNLL 2007 Shared Task.
  Prague, Czech Republic, 2007

- **[CL11]** Chih-Chung Chang and Chih-Jen Lin.
  LIBSVM: A library for support vector machines.
  ACM Transactions on Intelligent Systems and Technology, 2:27:1-27:27, 2011.
  Software available at `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

- **[CMP03]** Xavier Carreras, Lluıs Màrquez, and Lluís Padró.
  A simple named entity extractor using adaboost.
  In *Proceedings of CoNLL-2003 Shared Task*,
  Edmonton, Canada, June 2003.

- **[Fel98]** Christiane Fellbaum, editor.
  WordNet. An Electronic Lexical Database.
  Language, Speech, and Communication.
  The MIT Press, 1998.

- **[FSB+14]** Antske Fokkens, Aitor Soroa, Zuhaitz Beloki, Niels Ockeloen, German Rigau, Willem Robert van Hage, and Piek Vossen.
  NAF and GAF: Linking linguistic annotations.
  In *Proceedings 10th Joint ISO-ACL SIGSEM Workshop on Interoperable Semantic Annotation*, Reykjavik, Iceland, May 2014.

- **[KVHA95]** F. Karlsson, Atro Voutilainen, J. Heikkila, and A. Anttila, editors.
  Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text.
  Mouton de Gruyter, Berlin and New York, 1995.

- **[LCM13]** Xavier Lluís, Xavier Carreras, and Lluís Màrquez.
  Joint Arc-factored Parsing of Syntactic and Semantic Dependencies.
  In Transactions of the Association for Computational Linguistics (TACL), volume 1. Also presented at ACL-2013 (Sofia, Bulgaria).

- **[Pad98]** Lluís Padró.
  A Hybrid Environment for Syntax-Semantic Tagging.
  PhD thesis, Dept. Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya. February 1998.

- **[Sap13]** Emili Sapena, Lluís Padró, Jordi Turmo.
  A Constraint-Based Hypergraph Partitioning Approach to Coreference Resolution.
  Computational Linguistics vol. 39, n. 4, pg. 847-884. December, 2013.

- **[Vos98]** Piek Vossen, editor.
  EuroWordNet: A Multilingual Database with Lexical Semantic Networks.
  Kluwer Academic Publishers, Dordrecht, 1998.