FreeLing User Manual

3.1

October 2013

Contents

Chapter 1

Introduction

The FreeLing package consists of a library providing language analysis services (such as morphological analysis, date recognition, PoS tagging, 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, 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. Nevertheless, a sample main program is also provided as a basic interface to the library, which enables the user to analyze text files from the command line.

1.1 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.

1.2 What is NOT FreeLing

FreeLing is not a user-oriented text analysis tool. That is, it is not designed to be user friendly, 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 6 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.

1.3 Supported Languages

The current version supports (to different extents, see Table 1.3) Asturian (as), Catalan (ca), English (en), French (fr), Galician (gl), Italian (it), Portuguese (pt), Russian (ru), Slovene (sl), Spanish (es), and Welsh (cy).

	as	ca	cy	en	es	fr	gl	it	pt	ru	sl
Tokenization	X	X	X	X	X	X	X	X	X	X	
Sentence splitting	X	X	X	X	X	X	X	X	X	X	
Number detection		X		X	X		X	X	X	X	
Date detection		X		X	X		X		X	X	
Morphological dictionary	X	X	X	X	X	X	X	X	X	X	
Affix rules	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	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	X	
PoS tagging	X	X	X	X	X	X	X	X	X	X	
Phonetic encoding				X	X						
WN sense annotation		X		X	X		X				X
UKB sense disambiguation		X		X	X						X
Shallow parsing	X	X		X	X		X		X		
Full/dependency parsing	X	X		X	X		X				
Coreference resolution					X						

Table 1.1: Analysis services available for each language.

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.

1.4 License

FreeLing code is licensed under GNU General Public License (GPL).
The linguistic data collections are licensed 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.

1.5 Contributions

FreeLing is developed and maintained by people in TALP Research Center at Universitat Politecnica de Catalunya (http://www.talp.upc.edu).

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 (`http://nlp.lsi.upc.edu/freeling`).

Chapter 2

Getting it to work

2.1 Requirements

To install FreeLing you'll need:

- A typical Linux box with usual development tools:
  - bash
  - make
  - C++ compiler with basic STL support
- Enough hard disk space (1.3 Gb for source and compilation files, plus 600Mb for final installation)
- Some external libraries are required to compile FreeLing:
  - `libboost & libicu`
    Boost 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:
    * 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 (check the package manager in your system):

  * 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 2.2.

2.2 Installation

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

2.2.1 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 sections 2.2.2 and 2.2.3.

The provided packages will only work on debian-based distributions. They have been tested in Ubuntu (10.04LTS Lucid, 11.04 Natty, 12.04 Precise, 12.10 Quantal) and Debian (6.0.2 Squeeze, 7.0 Wheezy).

Most debian-bawsed 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 as described in section Section 2.1 to install all required packages.

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

2. Install freeling package:

   ```
   sudo dpkg -i freeling-3.0.deb
   ```

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

2.2.2 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:

   ```
   sudo apt-get install build-essential automake autoconf
   ```

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

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 2.1 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 (see section 2.1).

   Both in Debian (squeeze or wheezy) and in Ubuntu (Lucid or later) you need to do:

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

3. **Install FreeLing**

   ```
   tar xzvf freeling-3.0.tar.gz
   cd freeling-3.0
   ./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 sections 2.3 and 6 for details.

2.2.3 Install from SVN repositories

Installing from the SVN 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, and a SVN client.

   ```
   sudo apt-get install build-essential automake autoconf libtool subversion
   ```

   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 2.2.2 for this step.

3. **Checkout FreeLing sources**

   If you want the latest development version, do:

   ```
   svn checkout http://devel.cpl.upc.edu/freeling/svn/trunk myfreeling
   ```

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

   Alternatively, you can get any previous release with a command like:

   ```
   svn checkout http://devel.cpl.upc.edu/freeling/svn/versions/freeling-3.0-alfa1 myfreeling
   ```

(just replace `3.0-alfa1` with the right version number). You can see which versions are available by pointing your web browser to `http://devel.cpl.upc.edu/freeling/svn/versions`[1].

4. **Prepare local repositories for compilation**

```
cd myfreeling
aclocal; libtoolize; autoconf; automake -a
```

5. **Build and install FreeLing**

```
./configure
make
sudo make install
```

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

```
cd myfreeling
svn update
./configure
make
sudo make install
```

Depending on what changed in the repository, you may need to issue `aclocal; autoconf; automake -a` after `svn update`. You may also need to issue `make distclean` and repeat the process from `./configure` onwards.

2.2.4 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.

Most languages in FreeLing will work with this locale, but Russian will 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.

2.2.5 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.

- Download and install MacPorts following the instructions in `www.macports.org/install.php`. Note that you will need to install Apple XCode too, as described in the same page.

[1]Note that FreeLing-2.x versions require the appropiate `libomlet` and `libfries` versions (which you can checkout from the SVN in the same way). Check the *Download* section in FreeLing webpage to find out which *Omlet&Fries* version is suitable for each 2.x FreeLing version.

- Use MacPorts to install required developer tools:

  ```
  sudo port install automake
  sudo port install libtoool
  sudo port install subversion
  ```

- Use MacPorts to install required dependencies

  ```
  sudo port install boost
  ```

  This will install also libicu. Note that zlib is already installed in MacOS. If configure complains about it not being there, you can install it with `sudo port install zlib`.

- Compile and install FreeLing using the procedures described above (either "install from source" or "install from SVN"), but skipping the steps about installing development tools and dependencies.

  Important: libraries in MacOS are installed in `/opt/local` instead of `/usr/local`. So, when running `configure`, you need to specify the right library paths. Also, locales need some specific handling which requires the use of `libboost-locale`

  In summary, you need to run `./configure` with the command:

  ```
  ./configure --enable-boost-locale 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.

2.3 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 (e.g. customized input/output formats).

The usage and options of this main program is described in chapter 6.

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 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.

2.4 Porting to other platforms

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

Success have been reported on compiling FreeLing on MacOS, as well as on MS-Windows using cygwin (http://www.cygwin.com/).

Also, since version 3.0, project files are provided that allow to compile FreeLing under MSVC. Note that this files are contributor provided, and not supported by FreeLing developers. You'll find the files and a thorough README in the `msvc` folder inside FreeLing tarball.

You can visit the *Forum&FAQs* sections in FreeLing webpage for further help and details.

Chapter 3

Analysis 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 information and/or parameters needed by the module is 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, except the sentence splitter when used in non-flush mode (see section 3.3 below).

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.

3.1 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 &);
    /// load given language from given model file, add to existing languages.
    void add_language(const std::wstring&);
    /// train a model for a language, store in modelFile, and add
    /// it to the known languages list.
    void train_language(const std::wstring &, const std::wstring &,
                        const std::wstring &);
    /// Classify the input text and return the code of the best language (or "none")
    std::wstring identify_language (
                    const std::wstring&,
                    const std::set<std::wstring> &ls=std::set<std::wstring>()) const;
    /// fill a vector with sorted probabilities for each language
    void rank_languages (
              std::vector<std::pair<double,std::wstring> > &,
              const std::wstring &,
              const std::set<std::wstring> &ls=std::set<std::wstring>()) const;
};
```

Once created, the language identifier may be used to get the most likely language of a text (`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. The method `train_language` will use a plain text file 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.

3.1.1 Language Identifier Options File

The language identifier options file is divided in three sections: `<Languages>`, `<Threshold>`, and `<ScaleFactor>`, which are closed by `</Languages>`, `</Threshold>`, and `</ScaleFactor>`, respectively.

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.

Section `<Threshold>` and `<ScaleFactor>` contain one single line each, consisting of a real number in both cases.

The identifier uses a 4-gram visible Markov model to compute the probability of the text in each candidate language. Since the probabilitity of a sequence depends on its length, the result is divided by the text length to obtain a per-char "averaged" probability. Even in this way, the resulting probability is usually low and unintuitive. The parameter `ScaleFactor` multiplies this result to enlarge the difference between languages and to give probabilities in a more human scale. The parameter `Threshold` states minimun value that a language must achive to be considered a possible result. If no language reaches the threshold, the `identify_language` method will return `none`.

Note that this scaling is artificial, and doesn't change the results, only makes them more readable. The results with `ScaleFactor=1.0` and `Threshold=0.04` would be the same than with `ScaleFactor=5.0` and `Threshold=0.2`.

An example of a language identifier option file is:

```
<Languages>
./es.dat
./ca.dat
./it.dat
./pt.dat
</Languages>
<Threshold>
0.2
</Threshold>
<ScaleFactor>
5.0
</ScaleFactor>
```

3.2 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 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 &);

    /// tokenize string
    void tokenize(const std::wstring &, std::list<word> &) const;
    /// tokenize string, return result as list
    std::list<word> tokenize(const std::wstring &) const;
    /// tokenize string, tracking offset
    void tokenize(const std::wstring &, unsigned long &, std::list<word> &) const;
    /// tokenize string, tracking offset, return result as list
    std::list<word> tokenize(const std::wstring &, unsigned long &) 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

3.2.1 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.

3.3 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.

Note that the splitter is not thread-safe when the buffer is not flushed at each call.

The API for the splitter class is:

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

    /// Add list of words to the buffer, and return complete sentences
    /// that can be build.
    /// 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.
    std::list<sentence> split(const std::list<word> &, bool);
    /// split given list, add resulting sentence to output parameter
    void split(const std::list<word> &, bool, std::list<sentence> &);
};
```

3.3.1 Splitter Options File

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

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.

`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>
```

3.4 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 (from 3.5 to 3.13).

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.

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 &);

    /// 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 `maco_options` class has the following API:

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

    /// Submodules to activate
    bool UserMap, AffixAnalysis, MultiwordsDetection,
         NumbersDetection, PunctuationDetection,
         DatesDetection,   QuantitiesDetection,
         DictionarySearch, ProbabilityAssignment,
         NERecognition;

    /// Names of data files to provide to each submodule.
    std::string UserMapFile, LocutionsFile, QuantitiesFile,
            AffixFile, ProbabilityFile, DictionaryFile,
            NPdataFile, PunctuationFile;

    /// 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::string &);

    /// 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.
    /// The order of the parameters is the same than the variables defined above.
    void set_active_modules(bool,bool,bool,bool,bool,bool,bool,bool,bool,bool);
    void set_data_files(const std::wstring &,const std::wstring &,const std::wstring &,
                        const std::wstring &,const std::wstring &,const std::wstring &,
                        const std::wstring &,const std::wstring &);
    void set_nummerical_points(const std::string &,const std::string &);
    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 instantiate a `maco_options` object, initialize its fields with the desired values, and use it to call the constructor of the `maco` class.

The created object will create the required submodules, and when asked to `analyze` some sentences, it will just pass it down to each the submodule, and return the final result.

The `maco_options` class 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.

3.5 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 a finite-state automata 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 automata 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 &, const std::string &, 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 parameters that the constructor expects 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.

3.6 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: receives data file name
    punts(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 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.

3.6.1 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`

3.7 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. The API of the class is the following:

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

    /// 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 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 will 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.

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.

3.7.1 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.

3.8 Dates Detection Module

The dates detection module, as the number detection module in section 3.5, is a collection of language-specific finite-state automata, and for this reason needs no data file to be provided at instantiation time.

For languages that do not have a specific automata, 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 &);

    /// 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 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.

3.9 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 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 `-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 &, const std::wstring &,
               bool, const std::wstring &, bool invDic=false, bool retok=true);
    /// Destructor
    ~dictionary();

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

    /// Get dictionary entry for a given form, add to given analysis list.
    void search_form(const std::wstring &, std::list<analysis> &) const;
    /// Fills the analysis list of a word, checking for suffixes and contractions.
    /// Returns true iff the form is a contraction.
    bool annotate_word(word &, std::list<word> &, 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 &) const;
    /// Get possible forms for a lemma+pos (only if created with invDic=true)
    std::list<std::wstring> get_forms(const std::wstring &, const std::wstring &) const;

    /// 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 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.
- A boolean stating whether affixation analysis has to be applied.
- The affixation rules file name (it may be an empty string if the boolean above is set to false)

- An optional boolean (default: false) stating whether the dictionary must be created with inverse access, to enable the use of `get_forms` to obtain form given lemma+pos.
- 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.

3.9.1 Form Dictionary File

The form dictionary contais two required sections: `<IndexType>` and `<Entries>`, and two optional sections: `<LemmaPreferences>` and `<PosPreferences>`

Section `<IndexType>` conatins 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.

Optional sections `<LemmaPreferences>` and `<PosPreferences>` contain a list of pairs of lemmas or PoS tags, respectively. 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 asked 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).

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 (it may be it doesn't take into account the person as a feature), a preference is set for 3rd person (which is more frequent in standard text).

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.

Nevertheless, 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* $\rightarrow$ *dar+él*), derivation (e.g. *quick* $\rightarrow$ *quickly*, *rápida* $\rightarrow$ *rápidamente*), diminutive/augmentative sufixes, prefixes, or other similar behaviours, the affixation module should be used (see section 3.9.2 for details).

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).

3.9.2 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 is NCFS). 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` ot `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 it may have 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 VMIS1S0
```

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
```

3.9.3 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.[1] 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.

[1] Check the file `src/utilities/dicc-management/README` for details

3.10 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 &);

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

    /// 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;
};
```

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

3.10.1 Multiword Definition File

The file contains a list of multiwords to be recognized. The format of the file is one multiword 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
```

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`).

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.

3.11 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 WINDLL ner {
  public:
    /// Constructor
    ner(const std::wstring &);
    /// Destructor
    ~ner();

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

3.11.1 Basic NER module (`np`)

The first NER module is the `np` class, which is a just a FSA 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 &);

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

    /// 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 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, section 3.6) 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, the word *Spanish* in the sentence *He started studying Spanish two* *years ago* is not a 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 the sentences *whatever you say, I don't believe*, and *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 (*whatever you say, I don't believe*) unless some of its neighbour words are (*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 word *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.

```
<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:

```
<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 sentence `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 sentence `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).

- Section `<SplitMultiwords>` contains only one line with either `yes` or `no`. If `SplitMultiwords` is activated Named Entities still will be recognized but they will not be treated as a unit with only one Part-of-Speech tag for the whole compound. Each word gets its own Part-of-Speech tag instead.
  Capitalized words get the Part-of-Speech tag as specified in `NE_Tag`, The Part-of-Speech tags of non-capitalized words inside a Named Entity (typically, prepositions and articles) will be left untouched.

  ```
  <SplitMultiwords>
  no
  </SplitMultiwords>
  ```

3.11.2 *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, a simple 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 (in folder `src/utilities/nerc`). Check the README and comments in the scripts to find out what to do.

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 4.4.

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. These features are processed by `libfries`.

  ```
  <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 `libfries` to integer-encoded features needed by `libomlet`. The lexicon file is generated by `libfries` 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. 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 sentence `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 sentence `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).

- Section `<SplitMultiwords>` contains only one line with either `yes` or `no`. If `SplitMultiwords` is activated Named Entities still will be recognized but they will not be treated as a unit with only one Part-of-Speech tag for the whole compound. Each word gets its own Part-of-Speech tag instead.
  Capitalized words get the Part-of-Speech tag as specified in `NE_Tag`, The Part-of-Speech tags of non-capitalized words inside a Named Entity (typically, prepositions and articles) will be left untouched.

  ```
  <SplitMultiwords>
  no
  </SplitMultiwords>
  ```

3.12 Quantity Recognition Module

The `quantities` class is a FSA 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 (section 3.5). 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, a FSA 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 &, const std::string &);

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

    /// 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;
};
```

3.12.1 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 in multiwords file described in section 3.10, `code` is the type of magnitude the unit describes (currency, speed, etc.), and `tag` is a constraint on the lemmatized components of the multiword, following the same conventions than in multiwords file (section 3.10).

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 recognize strings 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 *There were many french francs*, the multiword won't be recognized since it is not assigning units to a determined quantity.

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.

3.13 Probability Assignment and Unkown 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 &, double);

    /// Assign probabilities for each analysis of given word
    void annotate_word(word &) const;
    /// Turn guesser on/of
    void set_activate_guesser(bool);

    /// 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 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`. 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.

3.13.1 Lexical Probabilities File

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` find out how to use it.

The probabilities file has nine sections: `<TagsetFile>`, `<UnknownTags>`, `<Theeta>`, `<Suffixes>`, `<SingleTagFreq>`, `<ClassTagFreq>`, `<FormTagFreq>`, `<BiassSuffixes>`, `<LidstoneLambda>`. Each section is closed by its corresponding tag `</TagsetFile>`, `</UnknownTags>`, `</Theeta>`, `</Suffixes>`, `</SingleTagFreq>`, `</ClassTagFreq>`, `</FormTagFreq>`, `</BiassSuffixes>`, `</LidstoneLambda>`.

- Section `<TagsetFile>`. This section contains a single line with the path to a tagset description file (see section 4.1) 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>`

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 `<LidstoneLambda>` specifies the λ parameter for Lidstone's Law smoothing.

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

  E.g.
  ```
  <LidstoneLambda>
  0.2
  </LidstoneLambda>
  ```

  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.

3.14 Alternatives 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 a correct word in the dictionary (*speed*), and return the correctly spelled alternatives. This module is based on the fast search algorithms on FSMs included in the finite-state libray FOMA (`http://code.google.com/p/foma`).

The API for this module is the following:

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

    /// direct access to results of underlying FSM
    void get_similar_words(const std::wstring &,
                           std::list<std::pair<std::wstring,int> > &) const;

    /// 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;
```

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

  This section must also contain one line
  `Dictionary filename`
  or either a line
  `PhoneticDictionary filename`
  stating the dictionary where the similar words are going to be searched. If `PhoneticDictionary` is stated, then an additional line
  `PhoneticRule filename`
  is expected, detailing the configuration file for a phonetic encoding module (see section 3.18) that will be used to encode the input forms before the search.

  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 (see section 3.9), since the morphological information will be ignored.

  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.

A line `CostMatrix filename` is expected, stating the 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).

A line `Threshold int-value` can be provided stating 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.

A line `MaxSizeDiff int-value` may also be provided. 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.

  The line `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).

  The line `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.

3.15 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 &);

    /// 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 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. See details in section 4.2.

- A section `<DataFiles>` containing at least the keyword `SenseDictFile` defined to a valid sense dictionary file name. E.g.:

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

The sense dictionary must follow the format described in section 4.2.2.

If the mapping rules `<WNposMap>` require a form dictionary, a keyword `formDictFile` with the dictionary to use must be provided in this section. More details are given in section 4.2.

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

3.16 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 described in section 3.15 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 (section 3.15) It ranks the candidate senses and sorts the list, according to the PageRank for each sense. The rank value is also provided in the result.

The API of the class is the following:

```
class ukb {
  public:
    /// Constructor. Receives a relation file for UKB, a sense dictionary,
    /// and two UKB parameters: epsilon and max iteration number.
    ukb(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 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>
  ```

  These parameters are UKB parameters: The an *epsilon* float value that controls the precision with with the end of PageRank iterations is decided, and a *MaxIterations* integer, that 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 one lines, 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.

  The *RelationFile* contains the semantic relationship graph to load. It is a text filecontaining pairs of related senses (WN synsets in this case). Relations are not labelled nor directed.

  An example of the content of this file is:

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

3.17 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:

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

    /// 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;

    /// given an analyzed sentence find out probability
    /// of the k-th best sequence
    double SequenceProb_log(const sentence &, 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.

The relax_tagger module can be tuned with hand written constraint, but is about 2 times slower than hmm_tagger. It is not able to produce k best sequences either.

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

    /// 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 `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 stating the maximum number of iterations to wait for convergence before stopping the disambiguation algorithm.
- A real number representing the scale factor of the constraint weights.
- A real number representing the threshold under which any changes will be considered too small. Used to detect convergence.
- 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.

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

3.17.1 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 script `src/utilities/train-tagger/bin/TRAIN.sh` provided in FreeLing package.
See `src/utilities/train-tagger/README` for details.

The file has eight sections: `<TagsetFile>`, `<Tag>`, `<Bigram>`, `<Trigram>`, `<Initial>`, `<Word>`, `<Smoothing>`, and `<Forbidden>`. Each section is closed by it corresponding tag `</Tag>`, `</Bigram>`, `</Trigram>`, etc.

The tag (unigram), bigram, and trigram probabilities are 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 (see section 4.1) 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 zero tag (for initial states) and for `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 zero indicates sentence-beggining.

  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`

  E.g. 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 zero indicates sentence-beggining.

  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`

  E.g. 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`

  E.g. 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" 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".

3.17.2 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 `label` with the `context`.
- `core` indicates the analysis or analyses (form interpretation) 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 woull to 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)`

  Conditions may be negated using the token `not`, i.e. `(not pos 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*(comió)` 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` states that the a match of the first term list is only acceptable if between the focus word and the matching word there is no match for the second term list.

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

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 word *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*);
```

3.18 Phonetic Encoding Module

The phonetic encoding module enriches words with their SAMPA[2] 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&);

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

  /// analyze given sentence, enriching words with phonetic encoding
  void analyze(sentence &) const;
  /// analyze given sentences
  void analyze(std::list<sentence> &) const;
  /// analyze sentence, return analyzed copy
  sentence analyze(const sentence &) const;
  /// analyze sentences, return analyzed copy
  std::list<sentence> analyze(const std::list<sentence> &) 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 with the method `word::get_ph_form`.

The format of the configuration file is the following:

There can be at most one exception dictionary, delimited by `<Exceptions>` and `</Exceptions>`. Each entry in the 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.

There can be 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.:

[2] www.phon.ucl.ac.uk/home/sampa/

```
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 are of the form `source/target/context`. Both `source` and `target` may be either a category name or a terminal string.

Simple string rules replace the first string with the second 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
```

First rule `qu/kw/_` replaces string `qu` with `kw` in any context.
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`, any character in category `V` and 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` and 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.

3.19 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`).

If you have an anotated corpus, the models can be trained using the scripts in `src/utilities/nerc`. See the README there and the comments inside the script for details.

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 4.4.

The API of the class is the following:

```
class nec {
  public:
    /// Constructor
    nec(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 constructor receives one parameter with the name of the configuration file for the module. Its content is described below.

3.19.1 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 `libfries` to integer-encoded features needed by `libomlet`. The lexicon file is generated by `libfries` 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>
```

3.20 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&);
   /// Get the start symbol of the grammar
   std::string get_start_symbol(void) const;

   /// 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 constructor receives a file with the CFG grammar to be used by the grammar, which is described in the next section

The method get_start_symbol returns the initial symbol of the grammar, and is needed by the dependency parser (see below).

3.20.1 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 (for instance: `VMIP3S0<"mylemmas.dat">`, or `VMI*("myforms.dat")`) it is interpreted as a file name, and the terminal will match any lemma (or word form) 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.

- `@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 "flat" non-terminal symbols 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 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...)

3.21 Dependency Parser Module

The Txala dependency parser [ACM05] gets parsed sentences –that is, `sentence` objects which have been enriched with a `parse_tree` by the `chart_parser` (or by any other means).

```
class dep_txala : public dependency_parser {
 public:
   /// constructor
   dep_txala(const std::string &, 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 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 in section 3.21.1.

3.21.1 Dependency Parsing Rule File

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

The file consists of four sections: sections: `<GRPAR>`, `<GRLAB>`, `<SEMDB>`, and `<CLASS>`, respectively closed by tags `</GRPAR>`, `</GRLAB>`, `</SEMDB>`, and `</CLASS>`.

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 contains a rule, with the format:

`priority flags context (ancestor,descendant) 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 `enabling_flags` equals "`-`", 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 ("`-`" means no limitations, and the rule is applied to any matching chunk pair) (see below).
- `(ancestor,descendant)` are the labels of the adjacent pair of chunks the rule will 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}`	any chunk labeled `np` by the chunker with a head word with a PoS tag matching the `^N.M` regular expression

- `operation` is the way in which `ancestor` and `descendant` nodes are to be combined (see below).
- The `op-params` component has two 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, and performing no relabelling of the new tree root.

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 rightmost chunk in the pair (`pp<of>`) under the leftmost (`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 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 npms[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 met. Then, the deepest rightmost node matching the label `npms[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

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 sibling have that path.

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).

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`.

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 4.2.

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.

```
mov     go      prep= to,towards
mov     come    prep= from
mov     walk    prep= through
mov     run     prep= to,towards   D.O.

animal "animals.dat"
```

3.22 Coreference Resolution Module

This module is a machine-learning based coreference solver, following the algorithm proposed by [SNL01]. It takes a document parsed by the shallow parser to detect noun phrases, and decides which noun phrases are coreferential.

The api of the module is the following:

```
class coref {
   public:
    /// Constructor
    coref(const std::string &, const int);

    /// Classify in coreference chains noun phrases in given document
    void analyze(document &) const;
};
```

The parameters received by the constructor are a filename, and an integer bitmask specifying which attributes have to be used by the classifier.

The meaning of the attributes can be found in the source file `include/freeling/coref_fex.h`. If you just want to use the module, set the value of this parameter to 0xFFFFFF to select all the attributes.

The string parameter is the name of the configuration file, which is described below:

3.22.1 Coreference Solver configuration file

The Coreference Solver module reads this file to find out some needed parameters. The file has three sections:

- Section `<ABModel>` path to the file containing the trained AdaBoost model. The `.abm` file contains an AdaBoost model 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 name of the file may be either absolute or relative to the location of the Coreference Solver config file.
  e.g:

```
<ABModel>
coref.abm
</ABModel>
```

It may be generated from an annotated corpus using the training programs that can be found in `src/utilities/coref`.

If you need to know more about this (e.g. to develop a Coreference Solver for your language) please contact FreeLing authors.

- Section `<SemDB>` specifies the files that contain a semantic database. This is required to compute some WN-based attributes used by the solver.

  The section must contain two lines specifying two semantic information files, a `SenseFile` and a `WNFile`. The filenames may be absolute or relative to the location of the dependency rules file. For example:

  ```
  <SEMDB>
  SenseFile ../senses30.src
  WNFile    ../../common/wn30.src
  </SEMDB>
  ```

- Section `<MaxDistance>` states the maximum distance (in words) at which possible corefernces will be considered. Short values will cause the solver to miss distant coreferents. Long distances will introduce a huge amount of possible coreferent candidate pairs, slow the system, and produce a larger amount of false positives.

Chapter 4

Other useful modules

FreeLing contains some internal classes used by the analysis modules described in Chapter 3.

Some applications may be interested in directly accessing these lower-level utilities, the most relevant of which are described in this chapter.

4.1 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 list of <feature,value> pairs with morphological
    /// information for given tag
    std::list<std::pair<std::wstring,std::wstring> >
              get_msf_features(const std::wstring &tag) const;

    /// get list <feature,value> pairs with morphological
    /// information, in a string format
    std::wstring get_msf_string(const std::wstring &tag) const;
};
```

The class constructor receives a file name with a tagset description. Format of the file is described below. The class offers two 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 depends on the language and part-of-speech, and 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.

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 list of string pairs, or as a formatted string.

4.1.1 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=sg` 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 tag digits. The form of each line is:

  ```
  tag short-tag-size digit-description-1 digit-description-2 ...
  ```

  where `tag` is the digit for the category in the EAGLES PoS tag (i.e. the first digit: `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 digits of the EAGLES PoS tag will we used as short version). Finally, fields `digit-description-n` contain information on how to interpret each digit in the EAGLES PoS tag

  There should be as many `digit-description` fields as digits there are in the PoS tag for that category. Each `digit-description` field has the format:
  `feature/digit:value;digit:value;digit:value;...`
  That is: the name of the feature encoded by that digit, followed by a slash, and then a semicolon-separated list of translation pairs that, for each possible digit in that position give the feature value.

  For instance, the rule for Spanish noun PoS tags is (in a single line):

  ```
  N 2 postype/C:common;P:proper gen/F:f;M:m;C:c num/S:s;P:p;N:c
                               neclass/S:person;G:location;O:organization;V:other
                               grade/A:augmentative;D:diminutive
  ```

  and states that any tag starting with `N` (unless it is found in section `<DirectTranslations>`) will be shortened using its two first digits (e.g. `NC`, or `NP`). Then, the description of each digit in the tag follows, encoding the information:

  1. `postype/C:common;P:proper` - second digit is the subcategory (feature `postype`) and its possible values are `C` (translated as `common`) and `P` (translated as `proper`).
  2. `gen/F:f;M:m;C:c` - third digit is the gender (feature `gen`) and its possible values are `F` (feminine, translated as `f`), `M` (masculine, translated as `m`), and `C` (common/invariable, translated as `c`).
  3. `num/S:s;P:p;N:c` - fourth digit is the number (feature `num`) and its possible values are `S` (singular, translated as `s`), `P` (plural, translated as `p`), and `C` (common/invariable, translated as `c`).

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/A:augmentative;D:diminutive` - sixth digit is the grade (feature `grade`) with possible values `A` (translated as `augmentative`), and `D` (translated as `diminutive`).

If a feature is underspecified or not appliable, a zero (`0`) is expected in the appropriate position of the PoS tag.

With the example rule described above, the tag translations in table 4.1.1 would take place:

EAGLES PoS tag	short version	morphological features
`NCMS00`	`NC`	`postype=common|gen=m|num=s`
`NCFC00`	`NC`	`postype=common|gen=f|num=c`
`NCFP0A`	`NC`	`postype=common|gen=f|num=p|grade=augmentative`
`NP0000`	`NP`	`postype=proper`
`NP00G0`	`NP`	`postype=proper|neclass=location`

Table 4.1: Example results for tagset managing module.

4.2 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 in section 4.2.1.
- 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 in section 4.2.2.

  `wnFile` is a file stating hyperonymy relations and other semantic information for each sense. The format is described in section 4.2.3.

  `formDictFile` may be needed if mapping rules in `<WNPosMap>` require it. It is a regular form file with morphological information, as described in section 3.9.

4.2.1 PoS mapping rules

Each line in section `<WNPosMap>` defines a mapping rule, with the 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`, and `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 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.

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.

4.2.2 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.

4.2.3 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 CyC 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.

4.3 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:
    /// build automaton from a file
    foma_FSM(const std::wstring &, const std::wstring &mcost=L"");
    /// delete FSM
    ~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 default cost for basic SED operations
    void set_basic_operation_cost(int);
  };
```

The constructor of the module requests one parameter stating the file to load, and a second optional parameter stating 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 `http://code.google.com/p/foma` 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 can 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.

4.4 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 used feature rules must follow the format described in section 4.4.1. 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 (mapping feature names to integer codes), and a map used to define custom feature functions, with associations between `feature_function` objects and function names (see section 4.4.5).

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.

4.4.1 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 in section 4.4.2

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 (see section 4.1) 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 (see section 4.4.4 below).

The `RULES` package starting keyword must be followed by a *condition* (see section 4.4.4).

The 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 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.

With this rule applied to the sentence *Hi ! , said John .*, the word *said* would get the features `punct_mark@-1`, `punct_mark@-2`, and `punct_mark@2`. 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).

4.4.2 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. See section 4.4.3 for a description of the syntax of feature name patterns.
- `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. See section 4.4.4 for a description of condition syntax.

4.4.3 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 avobe 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 (e.g. it will generate `punct_mark` twice for the word *John* in the avobe 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 (`$w(0)`) plus each possible PoS tag of the word right of it (`$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`, `pbig@-2:john_NNS`, `pbig@-1:lives_RB`, and `pbig@0:here_Fp`. 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,$2,...), then they are replaced with the strig matching the corresponding (sub)expression. This only makes sense if the condition of the rule included a regular expression match (see section 4.4.4). If it is not the case, results are undefined (probably a segmentation violation).
  2. If the bracket content is not a regex match variable, then it is interpreted as call to a custom feature function. It must have the format `{functname(position)}`, where `functname` is the name of the function as declared in the custom feature functions map (see section 4.4.5). The `position` parameter is the relative position to the focus word, and is interpreted in the same way than in the primitive features `$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.

4.4.4 Feature Rules Condition Syntax

Conditions control the applicability of a rule or a rule package.

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.

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 if any property in the list is found in the given file.
4. `<property-list> all_in_set <filename>` True if all properties in the list are found in the given file.
5. `<property-list> some_in_set <filename>` True if 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` true 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.

4.4.5 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. A map `std::map<std::wstring,const feature_function*>` needs to be given to the constructor, associating the custom function as 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 &, int, 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 a rule in the `.rgf` file refers to e.g. `{quote(0)}`.

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).

Chapter 5

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. Currently, the library provides a complete C++ API, a quite-complete Java API, and half-complete perl and python APIs.

5.1 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.

5.1.1 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 Data classes are defined in `libfries` library. Refer to the documentation in that library for the details on the classes.

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.

5.1.2 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).

5.2 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 very similar to the program depicted below, which reads text from stdin, morphologically analyzes it, and processes the obtained results.

Note that depending on the application, the input text could be obtained from a speech recognition 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() {
  wstring text;
  list<word> lw;
  list<sentence> ls;

  /// set locale to an UTF8 compatible locale
  util::init_locale(L"default");

  // if FreeLing was compiled with --enable-traces, you can activate
  // the required trace verbosity for the desired modules.
  //   traces::TraceLevel=4;
  //   traces::TraceModule=0xFFFFF;

  // ====== instantiate analyzers as needed =====

  wstring path=L"/usr/local/share/freeling/es/";
  tokenizer tk(path+L"tokenizer.dat");
  splitter sp(path+L"splitter.dat");

  // 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, set required options on/off
  opt.UserMap=false;                    opt.AffixAnalysis = true;
  opt.MultiwordsDetection = true;       opt.NumbersDetection = true;
  opt.PunctuationDetection = true;      opt.DatesDetection = true;
  opt.QuantitiesDetection = false;      opt.DictionarySearch = true;
  opt.ProbabilityAssignment = true;     opt.NERecognition = true;
  // alternatively, you can set active modules in a single call:
  // opt.set_active_modules(false, true, true, true, true, true, false, true, true, true);

  // and provide files for morphological submodules. Note that it is not necessary
  // to set opt.QuantitiesFile, since Quantities module was deactivated.
  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 can set the files in a single call:
  // opt.set_data_files(L"", path+L"locucions.dat", L"", path+L"afixos.dat",
  //                    path+L"probabilitats.dat", opt.DictionaryFile=path+L"maco.db",
  //                    path+L"np.dat", path+L"../common/punct.dat");

  // create the analyzer with the just build set of maco_options
  maco morfo(opt);
  // create a hmm tagger for spanish (with retokenization ability, and forced
  // to choose only one tag per word)
  hmm_tagger tagger(L"es", path+L"tagger.dat", true, true);
  // create chunker
  chart_parser parser(path+L"grammar-dep.dat");
  // create dependency parser
  dep_txala dep(path+L"dep/dependences.dat", parser.get_start_symbol());

  // ====== Start text processing =====

  // get plain text input lines while not EOF.
  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(lw, false);

    // perform  morphosyntactic analysis, disambiguation, and parsing
    morfo.analyze(ls);
    tagger.analyze(ls);
    parser.analyze(ls);
    dep.analyze(ls);

    // Do application-side processing with analysis results so far.
    ProcessResults(ls);

    // clear temporary lists;
    lw.clear(); ls.clear();
  }

  // No more lines to read. Make sure the splitter doesn't retain anything
  sp.split(lw, true, ls);

  // analyze sentence(s) which might be lingering in the buffer, if any.
  morfo.analyze(ls);
  tagger.analyze(ls);
```

```
  parser.analyze(ls);
  dep.analyze(ls);

  // process remaining sentences, if any.
  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.

Issuing `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.

Chapter 6

Using the sample main program to process corpora

The simplest way to use the FreeLing libraries is via the provided `analyzer` sample 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, the `analyzer` program 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 5.2

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, but if you don't need them, you can replace `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.

6.1 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 (`/usr/local/share/FreeLing/config` if you installed from source, and `/usr/share/FreeLing/config` if you used a binary `.deb` package).

Extra options may be specified in the command line to override any settings in `config-file`. See section 6.4.1 for details.

Server mode is triggered by option `--server`. See section 6.4.1 for details.

6.1.1 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.

The above command is equivalent to:

1. Define `FREELINGSHARE` variable to point to a directory with FreeLing data (e.g. `/usr/local/share/freeling`).
2. Make sure the `LD_LIBRARY_PATH` variable contains the directory where FreeLing libraries are installed (e.g. `/usr/local/lib`).
3. Locate the given configuration file in the current directory or in the FreeLing installation directories (e.g. `/usr/local/share/freeling/config`).
4. Run the `analyzer` (final `r`) program:
   `analyzer -f /usr/local/share/freeling/config/en.cfg <myinput >myoutput`

6.1.2 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 &
```

To launch the server, the script follows the same steps described in previous section, but with the options `--server --port 50005` passed to the final call to the `analyzer` executable.

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 6.4.1 for details on these options.

6.2 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.

6.3 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 6.4.2. Given the options there, the produced output would correspond to `morfo` format (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 --outf 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 --outf sense --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 --inpf morfo --outf tagged <mytext.mrf >mytext.tag
```

See options InputFormat and OutputFormat in section 6.4.1 for details on which are valid input and output formats.

6.4 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 6.4.1, both in their command-line and configuration file notations. Configuration files follow the usual linux standards. A sample file may be seen in section 6.4.2.

The 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.

6.4.1 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`

  Specify configuration file to use (default: analyzer.cfg).

- **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
Options management	0x00000008
Number detection	0x00000010
Date identification	0x00000020
Punctuation detection	0x00000040
Dictionary search	0x00000080
Affixation rules	0x00000100
Multiword detection	0x00000200
Named entity detection	0x00000400
Probability assignment	0x00000800
Quantities detection	0x00001000
Named entity classification	0x00002000
Automata (abstract)	0x00004000
Sense annotation	0x00010000
Chart parser	0x00020000
Parser grammar	0x00040000
Dependency parser	0x00080000
Correference resolution	0x00100000
Utilities	0x00200000
Word sense disambiguation	0x00400000
Ortographic correction	0x00800000
Database storage	0x01000000
Feature extraction	0x02000000
Language identifier	0x04000000
Omlet	0x08000000
Phonetics	0x10000000

- **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).

Other languages may be added to the library. See chapter 7 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
`--inpf <string>`	`InputFormat=<string>`

Format of input data (plain, token, splitted, morfo, tagged, sense).

  - 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`.
  - sense: tokenized, sentence-splitted, morphologically analyzed, PoS-tagged text, and sense-annotated. One token per line, sentences separated with one blank line.
    Each line has the format: `word (lemma tag prob sense`$_1$`:...:sense`$_N$`)`$^+$

- **Output Format**

Command line	Configuration file
`--outf <string>`	`OutputFormat=<string>`

Format of output data (token, splitted, morfo, tagged, shallow, parsed, dep).

  - 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)`$^+$
  or (if sense tagging has been activated):
  `word (lemma tag prob sense`$_1$`:...:sense`$_N$`)`$^+$
- 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 prob`
  or, if sense tagging has been activated: `word lemma tag prob sense`$_1$`:...:sense`$_N$
- shallow: tokenized, sentence-splitted, morphologically analyzed, PoS-tagged, optionally sense–annotated, and shallow-parsed text, as output 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 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.

- **Produce training output format**

Command line	Configuration file
`--train`	N/A

When this option (only available at command line) is specified, `OutputFormat` is forced to `tagged` and results are printed in the format:

```
word lemma tag # lemma1 tag1 lemma2 tag2 ...
```

that is, one word per line, with the selected lemma and tag as fields 2 and 3, a separator (`#`) and a list of all possible pairs lemma-tag for the word (including the selected one).

This format is expected by the training scripts. Thus, 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.

- **Language Identification Configuration File**

Command line	Configuration file
`-I <filename>`, `--fidn <filename>`	`N/A`

Configuration file for language identifier. See section 3.1 for details.

- **Tokenizer File**

Command line	Configuration file
`--abrev <filename>`	`TokenizerFile=<filename>`

File of tokenization rules. See section 3.2 for details.

- **Splitter File**

Command line	Configuration file
`--fsplit <filename>`	`SplitterFile=<filename>`

File of splitter options rules. See section 3.3 for details.

- **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. See section 3.9.2 for details.

- **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. See section 3.7 for details.

- **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. See section 3.10 for details.

- **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. See section 3.6 for details.

- **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. See section 3.12 for details.

- **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. See section 3.9 and chapter 7 for details.

- **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. See section 3.13 for details.

- **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 [bio|basic|none]`	`NERecognition=(bio|basic|none)`

Whether to perform NE recognition and which recognizer to use: "bio" for AdaBoost based NER, "basic" for a simple heuristic NE recognizer and "none" to perform no NE recognition . Deactivating this feature will cause the NE Classification module to have no effect.

- **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.

See section 3.11 for details.

- **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

See section 3.19 for details.

- **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

See section 3.18 for details.

- **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. See section 3.15 for details.

- **UKB Word Sense Disambiguator Configuration File**

Command line	Configuration file
`-U <filename>`, `--fukb <filename>`	`UKBConfigFile=<filename>`

  UKB configuration file. See section 3.16 for details.

- **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. See section 3.17.1 for details.

- **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. See section 3.17.2 for details.

- **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, see sections 3.9.2 and 3.17 for details.

- **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)

See 3.17 for more information.

- **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. See section 3.20.1 for details.

- **Dependency Parser Rule File**

Command line	Configuration file
`-T <filename>`, `--txala <filename>`	`DepTxalaFile==<filename>`

Rules to be used to perform dependency analysis. See section 3.21.1 for details.

- **Coreference Resolution**

Command line	Configuration file
`--coref`, `--nocoref`	`CoreferenceResolution=(yes|y|on|no|n|off)`

Whether to perform coreference resolution.

- **Named Entity Classifier File**

Command line	Configuration file
`-C <filename>`, `--fcorf <filename>`	`CorefFile=<filename>`

  Configuration file for coreference resolution module.

  See section 3.22 for details.

6.4.2 Sample Configuration File

A sample configuration file follows. This is only a sample, and probably won't work if you use it as is. You can start using freeling with the default configuration files which –after installation– are located in `/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
##

TraceLevel=3
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.
## Valid input formats are: plain, token, splitted, morfo, tagged, sense.
## Valid output formats are: : plain, token, splitted, morfo, tagged,
## shallow, parsed, dep.
InputFormat=plain
OutputFormat=tagged

# 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
MultiwordsDetection=yes
NumbersDetection=yes
PunctuationDetection=yes
DatesDetection=yes
QuantitiesDetection=yes
DictionarySearch=yes
ProbabilityAssignment=yes
OrthographicCorrection=no
DecimalPoint=,
ThousandPoint=.
LocutionsFile=$FREELINGSHARE/es/locucions.dat
QuantitiesFile=$FREELINGSHARE/es/quantities.dat
AffixFile=$FREELINGSHARE/es/afixos.dat
ProbabilityFile=$FREELINGSHARE/es/probabilitats.dat
```

```
DictionaryFile=$FREELINGSHARE/es/dicc.src
PunctuationFile=$FREELINGSHARE/common/punct.dat
ProbabilityThreshold=0.001

# NER options
NERecognition=yes
NPDataFile=$FREELINGSHARE/es/np.dat
## comment line above and uncomment that below, if you want
## a better NE recognizer (higer accuracy, lower speed)
#NPDataFile=$FREELINGSHARE/es/ner/ner-ab.dat

#Spelling Corrector config file
CorrectorFile=$FREELINGSHARE/es/corrector/corrector.dat

## Phonetic encoding of words.
Phonetics=no
PhoneticsFile=$FREELINGSHARE/es/phonetics.dat

## NEC options
NEClassification=no
NECFile=$FREELINGSHARE/es/nec/nec-svm.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.dat
TaggerRelaxMaxIter=500
TaggerRelaxScaleFactor=670.0
TaggerRelaxEpsilon=0.001
TaggerRetokenize=yes
TaggerForceSelect=tagger

#### Parser options
GrammarFile=$FREELINGSHARE/es/grammar-dep.dat

#### Dependence Parser options
DepTxalaFile=$FREELINGSHARE/es/dep/dependences.dat

#### Coreference Solver options
CoreferenceResolution=no
CorefFile=$FREELINGSHARE/es/coref/coref.dat
```

Chapter 7

Extending the library with analyzers for new languages

It is possible to extend the library with capability to deal with a new language. In some cases, this may be done without reprogramming, but for accurate results, some modules would 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, such as the example presented in section 6.4.2).

7.1 Tokenizer

The first module in the processing chain is the tokenizer. As described in section 6.4.1, 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.

7.2 Morphological analyzer

The morphological analyzer module consists of several sub-modules that may require language customization. See sections 3.4 to 3.13 for details on data file formats for each module:

7.2.1 Multiword detection

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.

7.2.2 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`.

7.2.3 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 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 *dates_es*, *dates_en*, and *dates_ca* classes. State/transition diagrams of those automata can be found in the directory `doc/diagrams`.

7.2.4 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 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 *quantities_es* and *quantities_ca* 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 CurrencyFile 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 CurrencyFile, 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 CurrencyFile contents. Since that file will be used only for your language, feel free to readjust its format.

7.2.5 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 `indexdict` program you'll find in the binaries directory of FreeLing. This program reads data from stdin and indexes them into a DB file with the name given as a parameter.

The input data is expected to contain one word form per line, each line with the format:

```
form lemma1 tag1 lemma2 tag2 ...
```

E.g.

```
abalanzará abalanzar VMIC1S0 abalanzar VMIC3S0
bajo bajar VMIP1S0 bajo AQ0MS0 bajo NCMS000 bajo SPS00
efusivas efusivo AQ0FP0
```

7.2.6 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. See section 3.9.2 for details on affixation rules format.

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 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.

7.2.7 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.

See section 3.13 for format details.

7.3 HMM PoS Tagger

The HMM PoS tagger only requires an appropriate HMM parameters file, given by the TaggerHMMFile option. See section 3.17.1 for format details.

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.

7.4 Relaxation Labelling PoS Tagger

The Relaxation Labelling PoS tagger only requires an appropriate pseudo- constraint grammar file, given by the RelaxTaggerFile option. See section 3.17.2 for format details.

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 an annotated corpus) is using the estimation and smoothing script `src/utilities/train-tagger/bin/TRAIN.sh` provided in FreeLing source package.

The produced constraint grammar files contain only simple bigram constraints, but the model may be improved by hand coding more complex context constraint, as can be seen in the Spanish data file in `share/FreeLing/es/constr_grammar.dat`

7.5 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.

A README file and training scripts can be found in `src/utilities/nerc` in FreeLing tarball.

7.6 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). See section 3.20.1 for format details.

7.7 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). See section 3.21.1 for format details.

Bibliography

[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*. ACL, 2000.

[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 `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[CMP03] Xavier Carreras, Lluís 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.

[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.

[Pad98] Lluís Padró. *A Hybrid Environment for Syntax–Semantic Tagging.* PhD thesis, Dept. Llenguatges i Sistemes Informàtics. Universitat Politècnica de Catalunya, February 1998. http://www.lsi.upc.edu/~padro.

[SNL01] W. M. Soon, H. T. Ng, and D. C. Y. Lim. A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics*, 27(4):521–544, 2001.

[Vos98] Piek Vossen, editor. *EuroWordNet: A Multilingual Database with Lexical Semantic Networks.* Kluwer Academic Publishers, Dordrecht, 1998.